

Fig. 1a

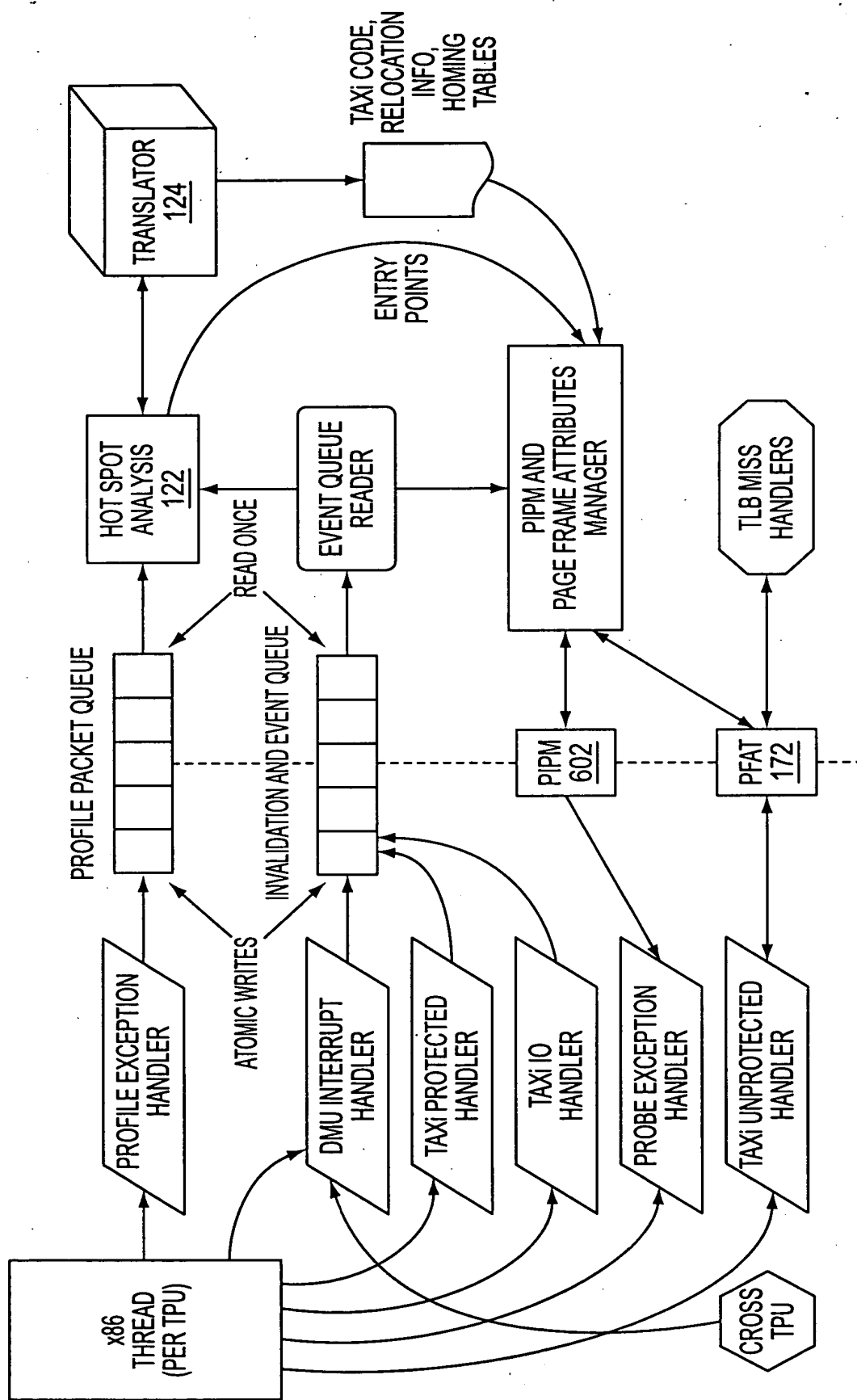


FIG. 1B

FIG. 1C

The diagram illustrates a processor architecture 100, organized into three main functional regions: MEMORY UNIT, GBUS, and DATAPATH.

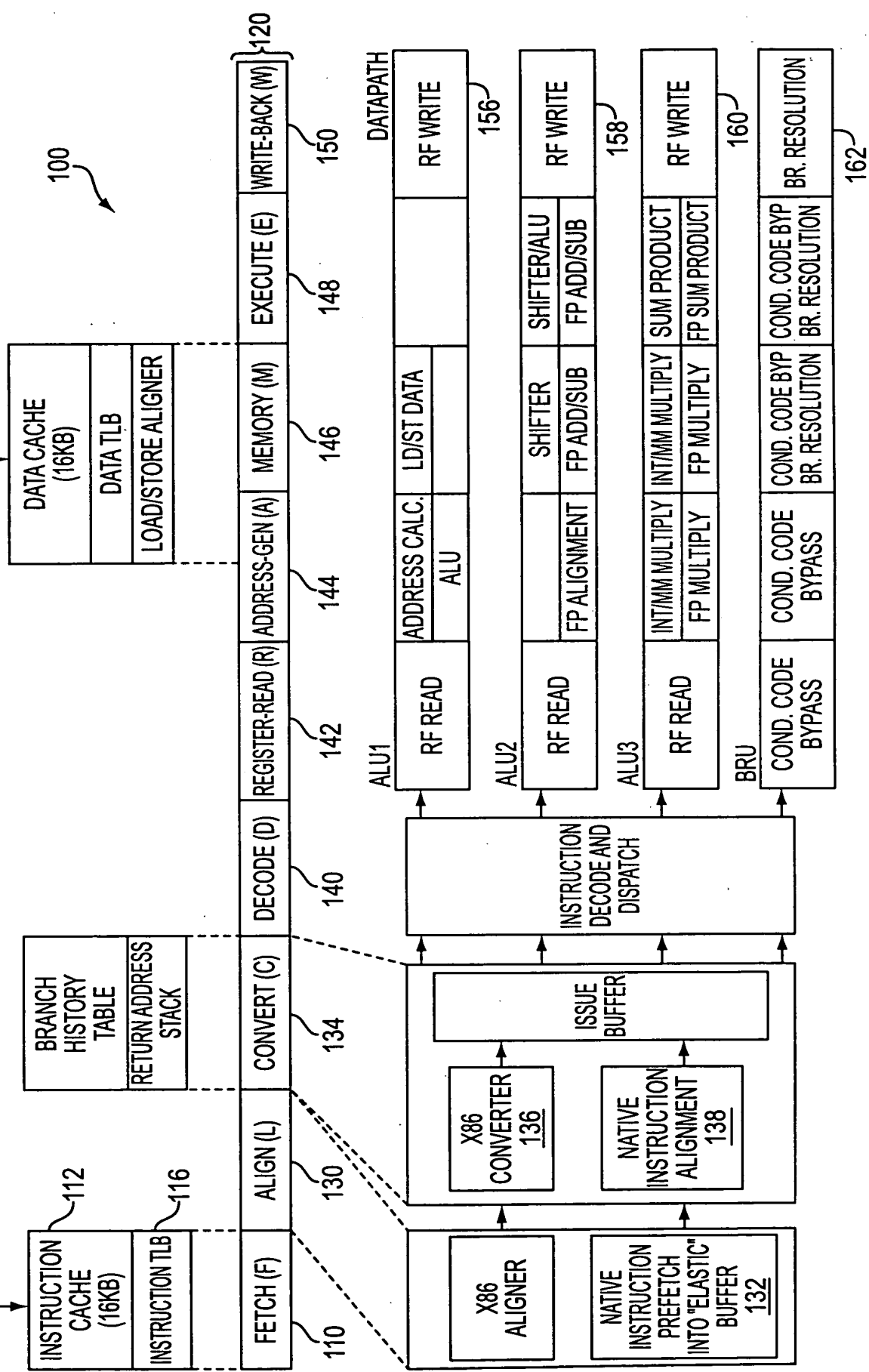
MEMORY UNIT: This region contains the INSTRUCTION CACHE (16KB) and INSTRUCTION TLB (112), the DATA CACHE (16KB) and DATA TLB (116), and the LOAD/STORE ALIGNER (110). It is connected to the GBUS via bidirectional arrows.

GBUS: The central communication bus, represented by a horizontal line with arrows at both ends, connecting the memory unit to the datapath.

DATAPATH: This region is divided into several stages and functional blocks:

- FETCH (F) 110:** The initial stage, which includes an X86 ALIGNER and a NATIVE INSTRUCTION PREFETCH INTO "ELASTIC" BUFFER 132.
- ALIGN (L) 130:** Contains an X86 CONVERTER 136 and a NATIVE INSTRUCTION ALIGNMENT 138.
- CONVERT (C) 134:** Includes a BRANCH HISTORY TABLE and a RETURN ADDRESS STACK.
- DECODE (D) 140:** The INSTRUCTION DECODE AND DISPATCH block, which receives input from the CONVERT stage and feeds into the ALU stages.
- REGISTER-READ (R) 142:** The first ALU stage (ALU1), containing an RF READ and an ADDRESS CALC. ALU.
- ADDRESS-GEN (A) 144:** The second ALU stage (ALU2), containing an RF READ, a SHIFTER, and FP ALIGNMENT/SUB.
- MEMORY (M) 146:** The third ALU stage (ALU3), containing an RF READ, INT/MM MULTIPLY, FP MULTIPLY, and SUM PRODUCT/FP SUM PRODUCT.
- EXECUTE (E) 148:** The final ALU stage (BRU), containing a COND. CODE BYPASS, COND. CODE BYPASS, COND. CODE BYPASS, and BR. RESOLUTION.
- WRITE-BACK (W) 150:** The final stage of the datapath.

Connections are shown with solid and dashed lines, indicating data and control paths between the various components.



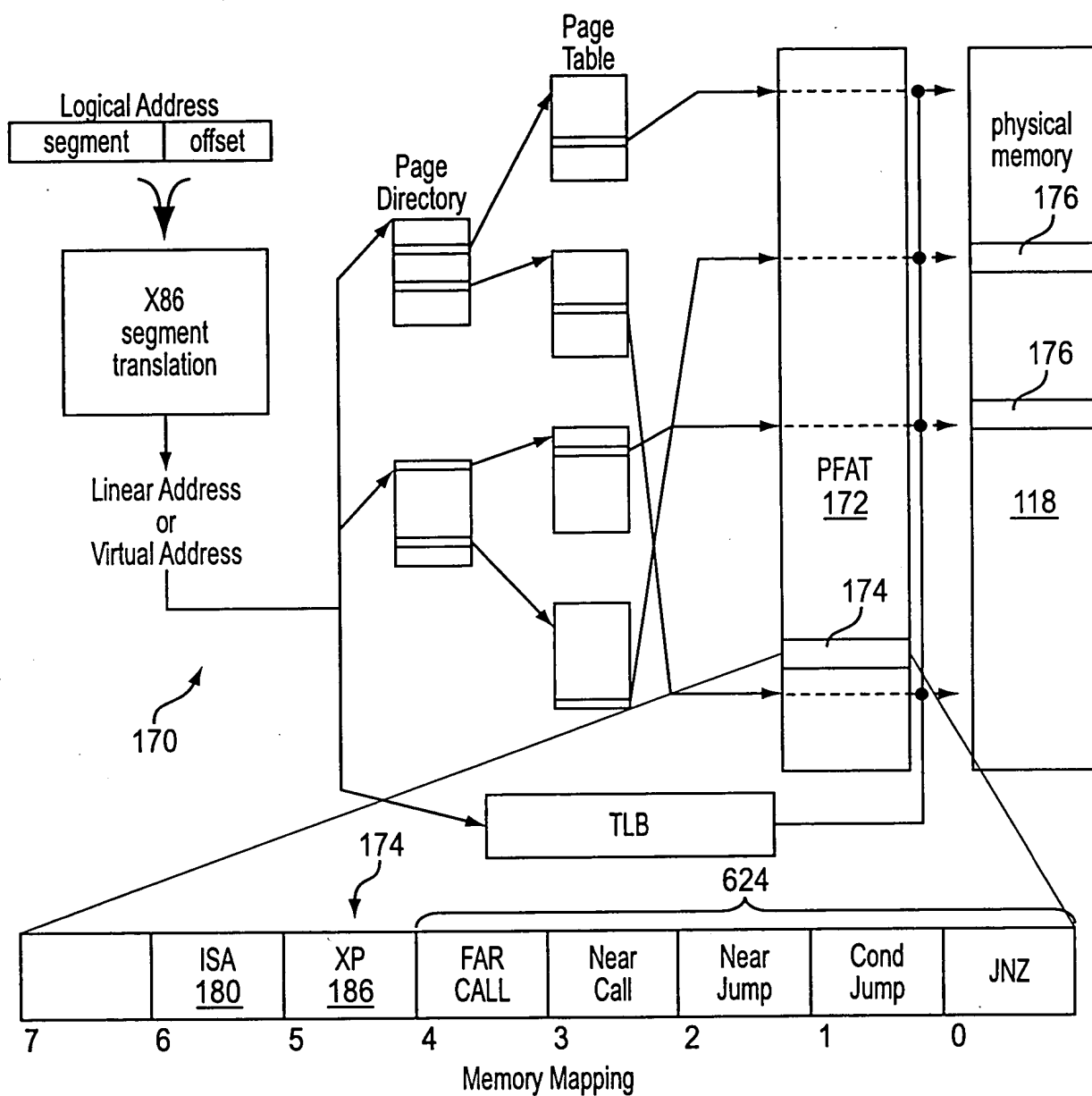


FIG. 1D

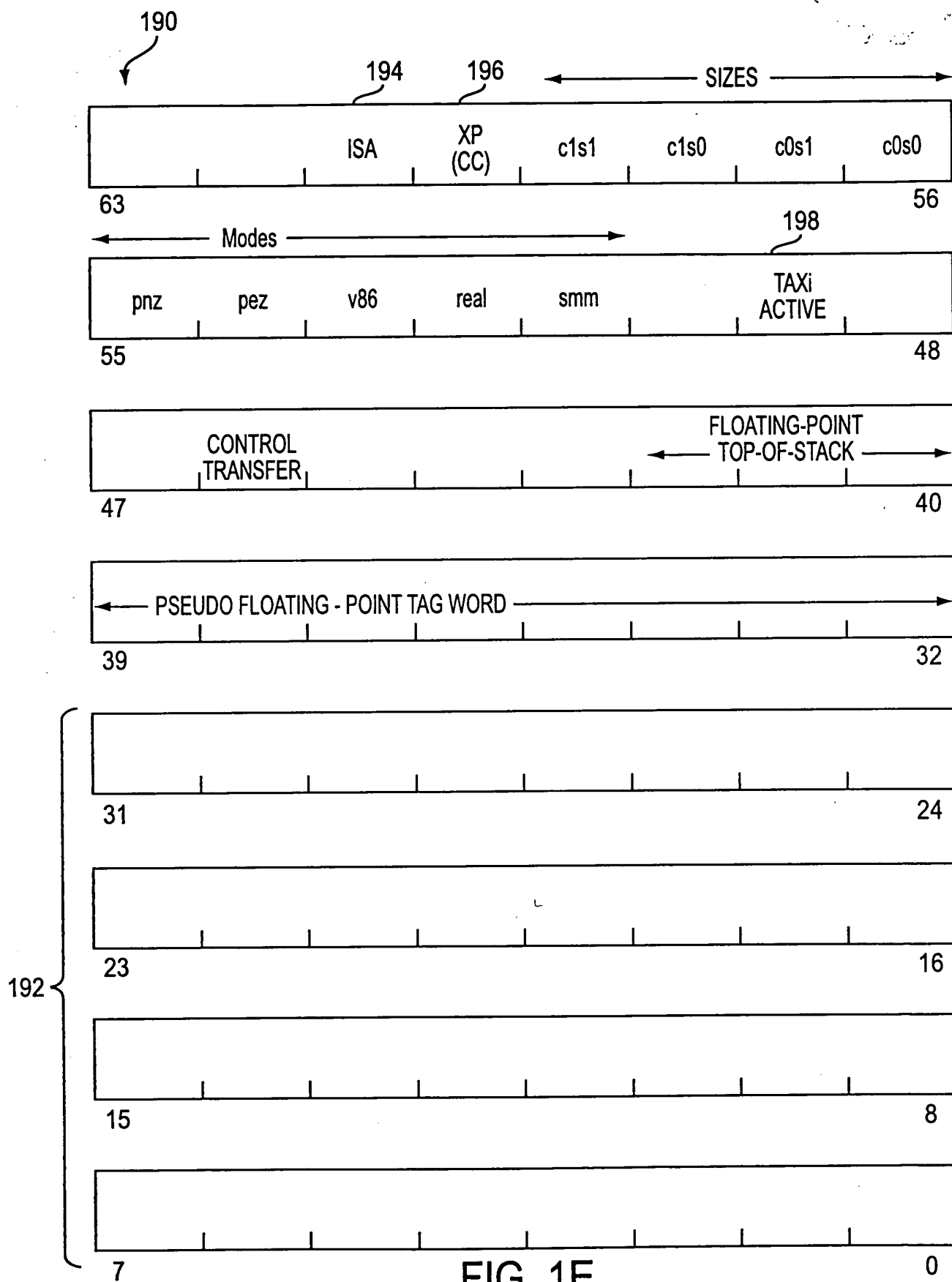


FIG. 1E

I-TLB PROPERTY BITS	DECODED PROPERTY VALUES			PROTECTED INTERPRETATION	INSTRUCTIONS SENT TO:	COLLECT PROFILE TRACE- PACKETS?	PROBE FOR TRANSLATED CODE	I/O MEMORY REFERENCE EXCEPTIONS
	ISA 194	CC 200						
00	TAP	TAP	NO	NATIVE CODE OBSERVING NATIVE RISCy CALLING CONVENTIONS	NATIVE DECODER	NO	NO	FAULT IF SEG.tio
01	TAP	x86	NO	NATIVE CODE OBSERVING x86 CALLING CONVENTIONS	NATIVE DECODER	NO	NO	FAULT IF SEG.tio
10	x86	x86	NO	x86 CODE, UNPROTECTED - TAX! PROFILE COLLECTION ONLY	x86 HW CONVERTER	IF ENABLED	NO	TRAP IF PROFILING
11	x86	x86	YES	x86 CODE, PROTECTED - TAX! CODE MAY BE AVAILABLE	x86 HW CONVERTER	IF ENABLED	BASED ON I-TLB PROBE ATTRIBUTES	TRAP IF PROFILING

180,182,
184,186

184,186

FIG. 2A

TRANSITION (SOURCE => DEST) ISA & CC PROPERTY VALUES		HANDLER ACTION
212	00 => 00	NO TRANSITION EXCEPTION
214	00 => 01	VECT_XXX_X86_CC EXCEPTION - HANDLER CONVERTS FROM NATIVE TO x86 CONVENTIONS
216	00 => 1x	VECT_XXX_X86_CC EXCEPTION - HANDLER CONVERTS FROM NATIVE x86 CONVENTIONS, SETS UP EXPECTED EMULATOR AND PROFILING STATE
218	01 => 00	VECT_XXX_TAP_CC EXCEPTION - HANDLER CONVERTS FROM x86 TO NATIVE CONVENTIONS
220	01 => 01	NO TRANSITION EXCEPTION
222	01 => 1x	VECT_X86_ISA EXCEPTION [CONDITIONAL BASED ON PCW.X86_ISA_ENABLE FLAG] - SETS UP EXPECTED EMULATOR AND PROFILING STATE
224	1x => 00	VECT_XXX_TAP_CC EXCEPTION - HANDLER CONVERTS FROM x86 TO NATIVE CONVENTIONS
226	1x => 01	VECT_TAP_ISA EXCEPTION [CONDITIONAL BASED PCW.TAP_ISA_ENABLE FLAG] - NO CONVENTION CONVERSION NECESSARY
228	1x => 10	NO TRANSITION EXCEPTION - [PROFILE COMPLETE POSSIBLE, PROBE POSSIBLE]
230	1x => 11	NO TRANSITION EXCEPTION - [PROFILE COMPLETE POSSIBLE, PROBE NOT POSSIBLE]

FIG. 2B

NAME	DESCRIPTION	TYPE
242 VECT_call_X86_CC	PUSH ARGS, RETURN ADDRESS, SET UP x86 STATE	FAULT ON TARGET INSTRUCTION
244 VECT_jump_X86_CC	SET UP x86 STATE	FAULT ON TARGET INSTRUCTION
246 VECT_ret_no_fp_X86_CC	RETURN VALUE TO EAX:EDX, SET UP x86 STATE	FAULT ON TARGET INSTRUCTION
248 VECT_ret_fp_X86_CC	RETURN VALUE TO x86 FP STACK, SET UP x86 STATE	FAULT ON TARGET INSTRUCTION
250 VECT_call_TAP_CC	x86 STACK ARGS, RETURN ADDRESS TO REGISTERS	FAULT ON TARGET INSTRUCTION
252 VECT_jump_TAP_CC	x86 STACK ARGS TO REGISTERS	FAULT ON TARGET INSTRUCTION
254 VECT_ret_no_fp_TAP_CC	RETURN VALUE TO RVO	FAULT ON TARGET INSTRUCTION
256 VECT_ret_any_TAP_CC	RETURN TYPE UNKNOWN, SETUP RVO AND RVDP	FAULT ON TARGET INSTRUCTION

FIG. 2C

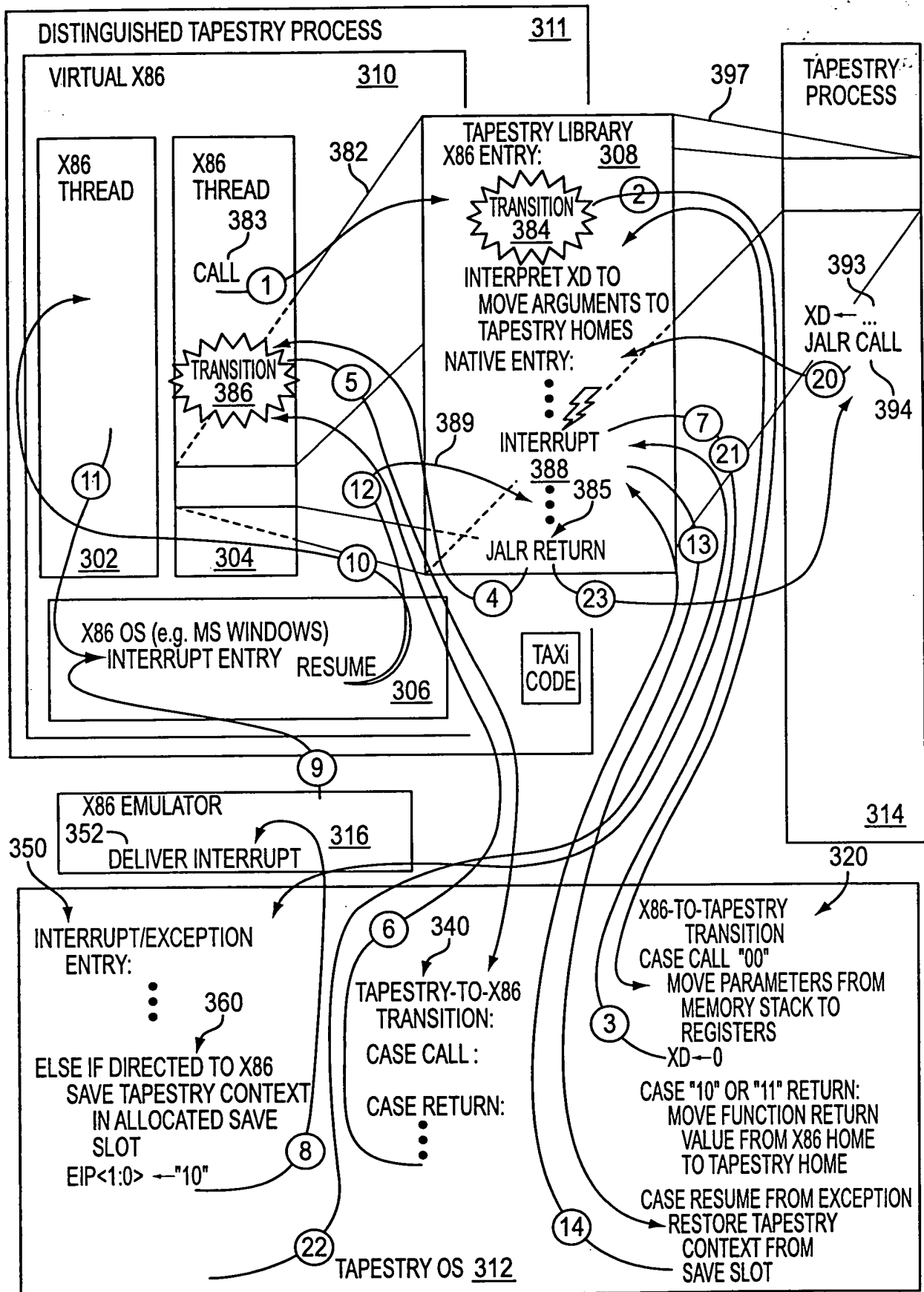


FIG. 3A

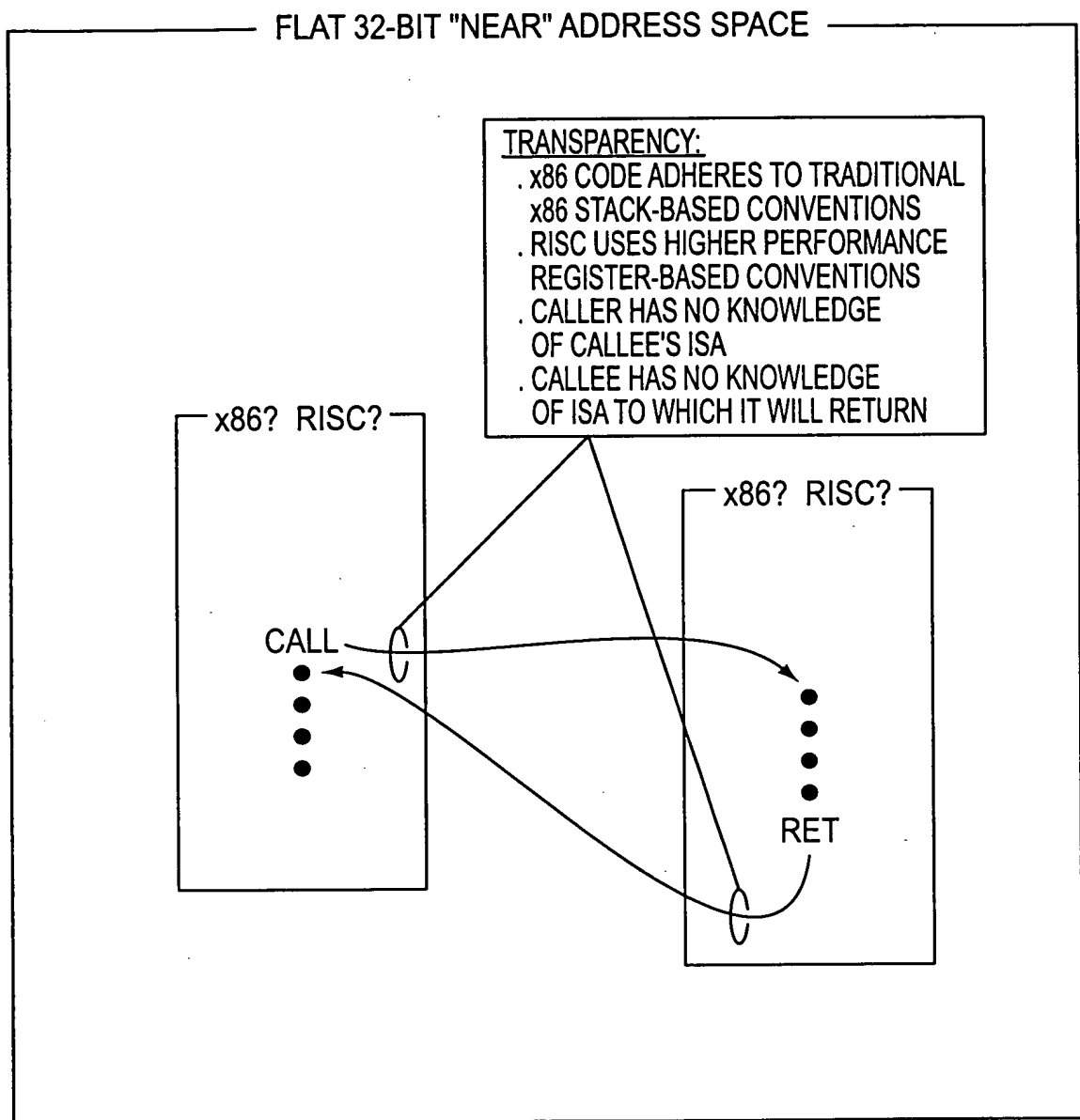


FIG. 3B

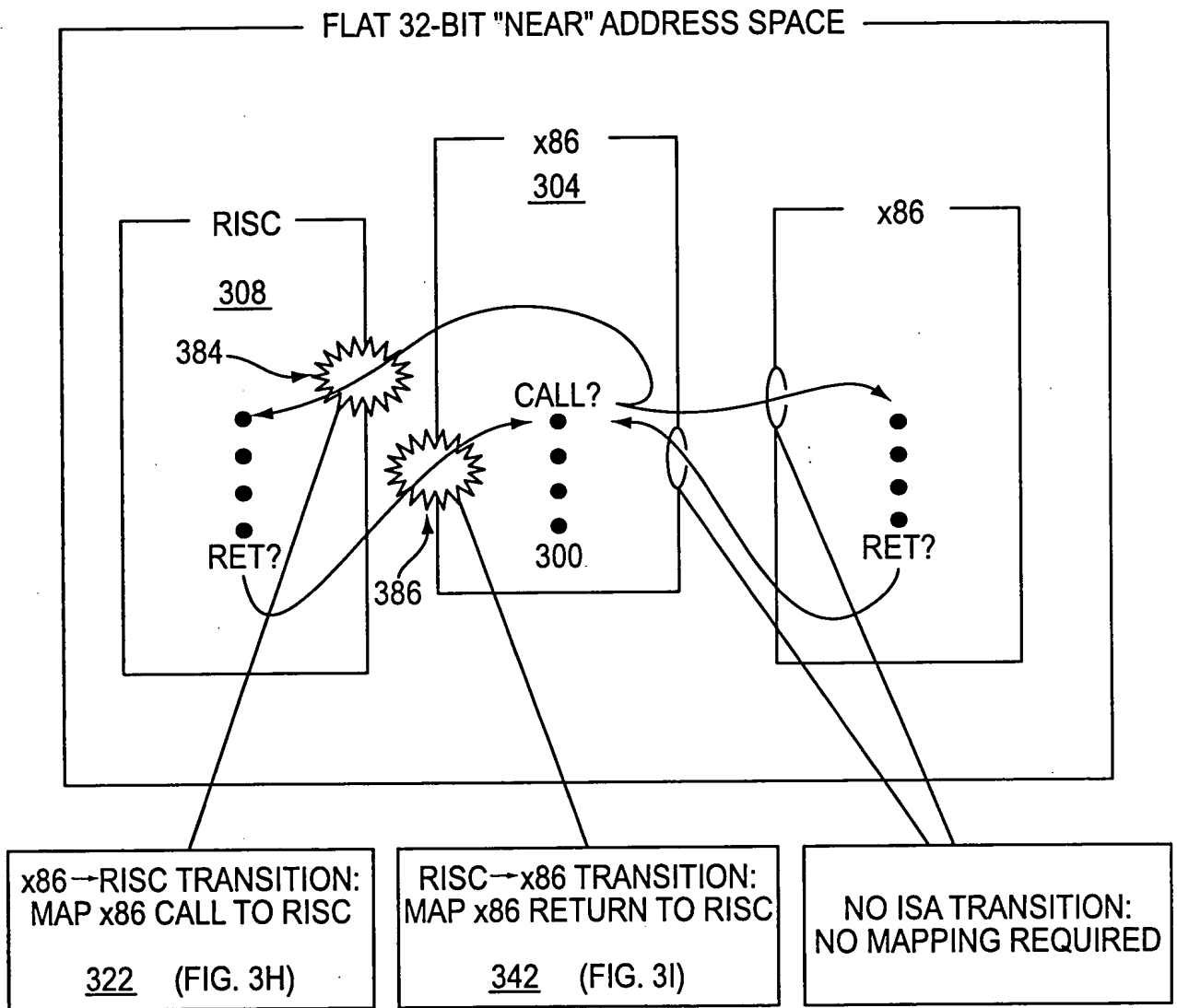


FIG. 3C

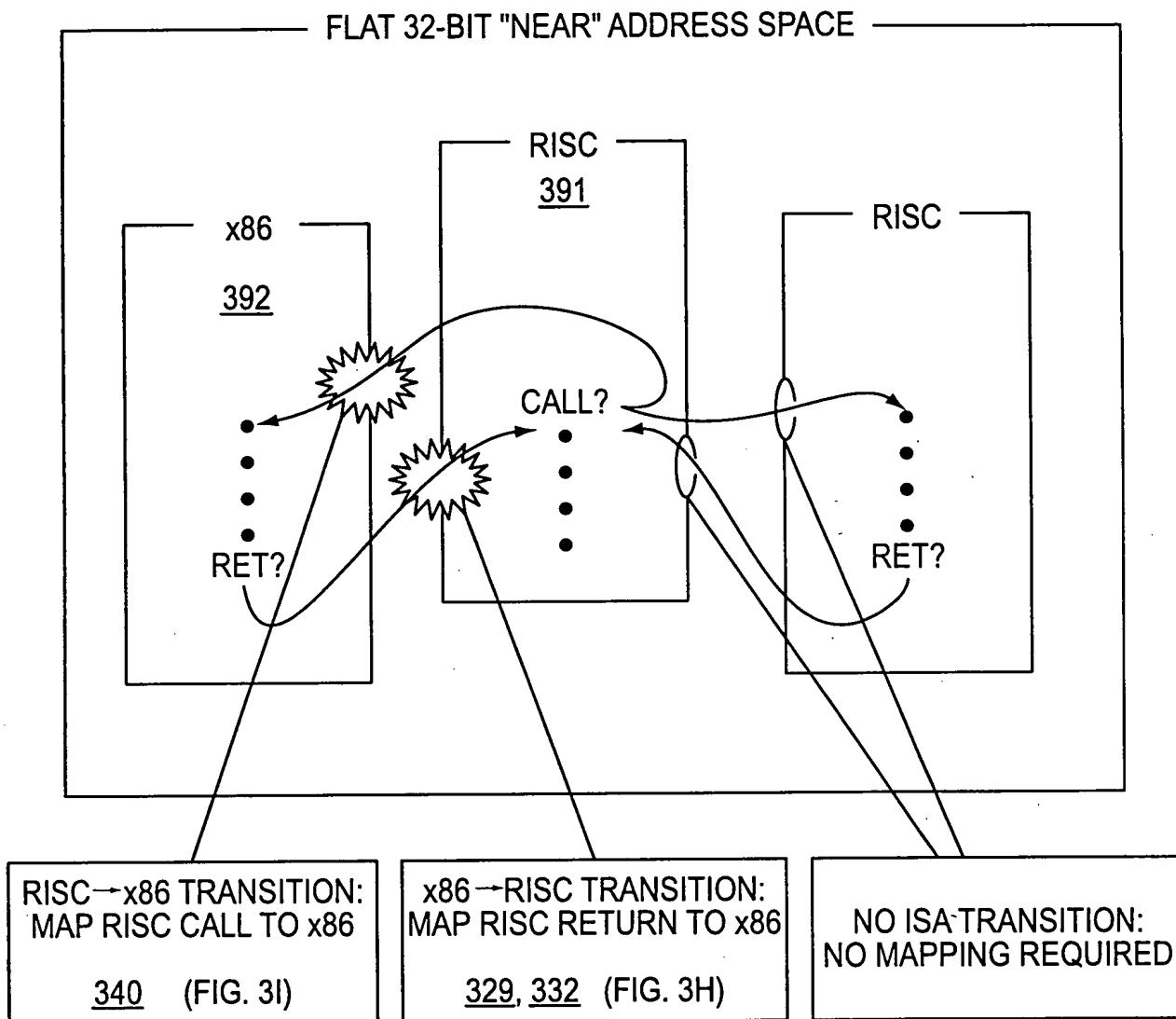


FIG. 3D

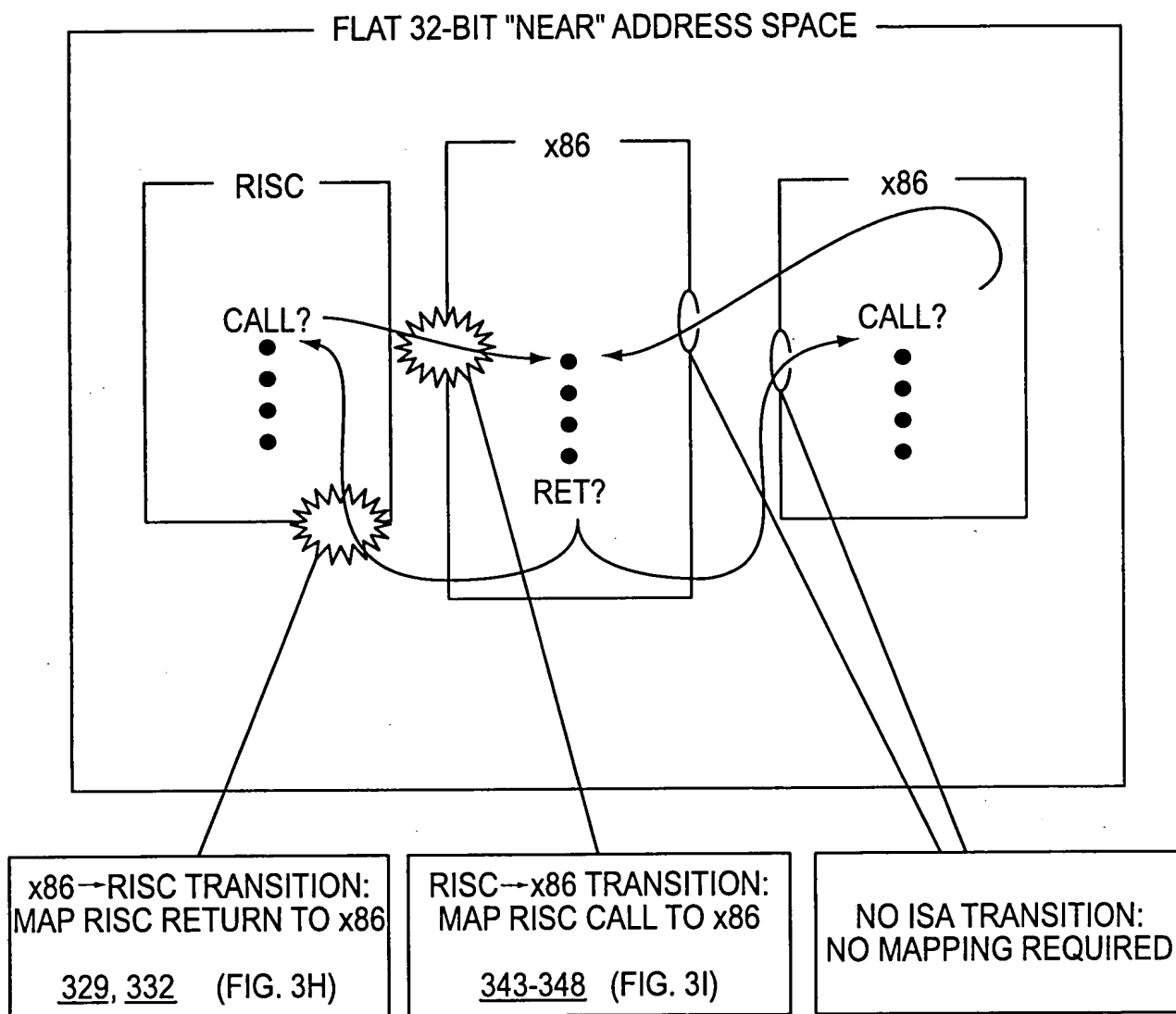


FIG. 3E

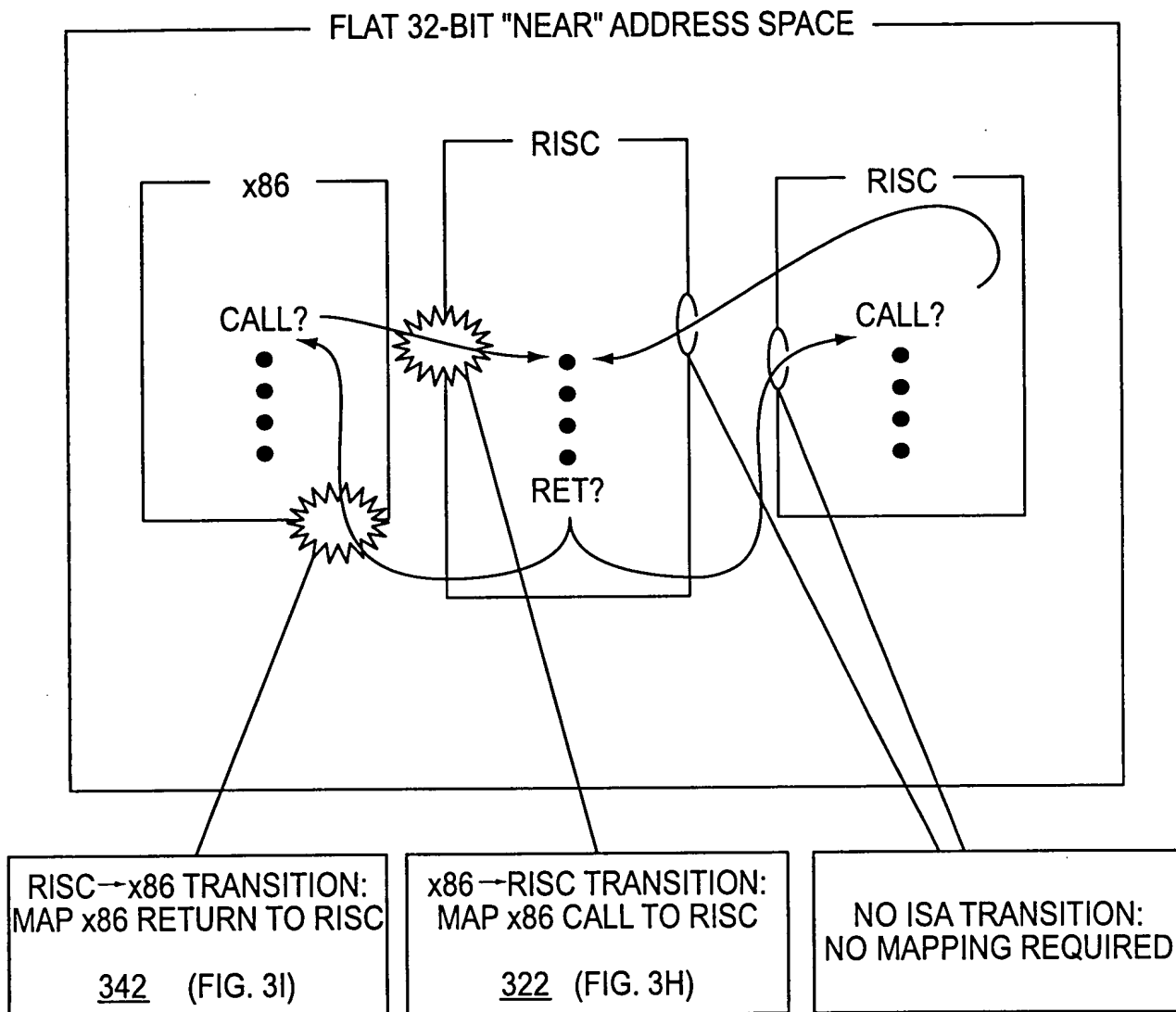


FIG. 3F

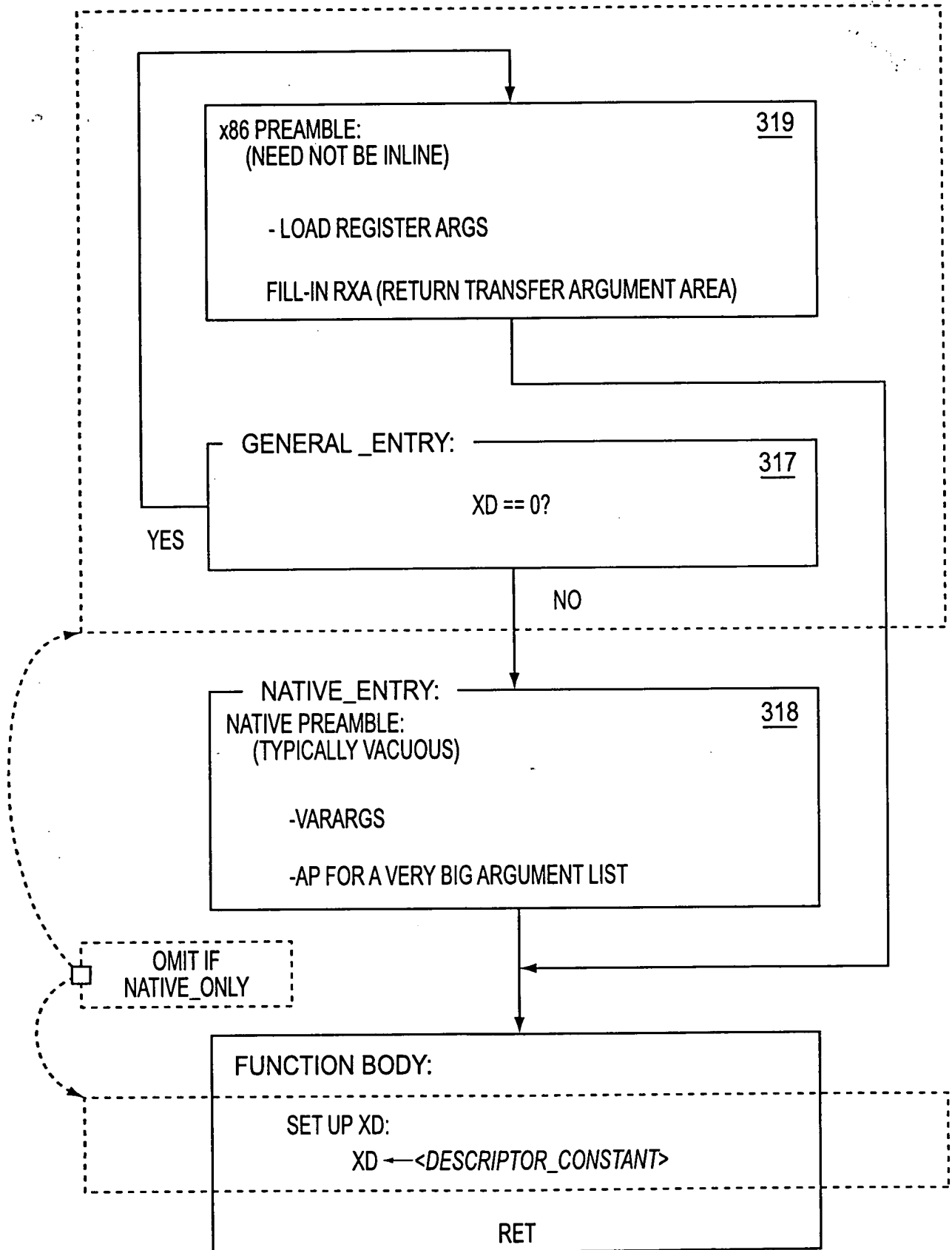


FIG. 3G

X86-to Tapestry transition exception handler

// This handler is entered under the following conditions:

// 1. An x86 caller invokes a native function

// 2. An x86 function returns to a native caller

// 3. x86 software returns to or resumes an interrupted native function following

// an external asynchronous interrupt, a processor exception, or a context switch

```

    dispatch on the two least-significant bits of the destination address {
    case "00" // calling a native subprogram
        // copy linkage and stack frame information and call parameters from the memory
        // stack to the analogous Tapestry registers
        LR ← [SP++] // set up linkage register 323
        AP ← SP // address of first argument 324
        SP ← SP - 8 // allocate return transfer argument area 326
        SP ← SP & (-32) // round the stack pointer down to a 0 mod 32 boundary 327
        XD ← 0 // inform callee that caller uses X86 calling conventions 328
    case "01" // resuming an X86 thread suspended during execution of a native routine
        if the redundant copies of the save slot number in EAX and EDX do not match or if
            the redundant copies of the timestamp in EBX:ECX and ESI:EDI do not match { 371
            // some form of bug or thread corruption has been detected
            goto TAPESTRY_CRASH_SYSTEM( thread-corruption-error-code ) 372
        }
        save the EBX:ECX timestamp in a 64-bit exception handler temporary register 373
        (this will not be overwritten during restoration of the full native context)
        use save slot number in EAX to locate actual save slot storage 374
        restore full entire native context (includes new values for all x86 registers) 375
        if save slot's timestamp does not match the saved timestamp { 376
            // save slot has been reallocated; save slot exhaustion has been detected
            goto TAPESTRY_CRASH_SYSTEM( save-slot-overwritten-error-code ) 377
        }
        free the save slot 378
    case "10" // returning from X86 callee to native caller, result already in registers
        RV0<63:32> ← edx<31:00> // in case result is 64 bits 333
        convert the FP top-of-stack value from 80 bit X86 form to 64-bit form in RVDP 334
        SP ← ESI // restore SP from time of call 337
    case "11" // returning from X86 callee to native caller, load large result from memory
        RV0..RV3 ← load 32 bytes from [ESI-32] // (guaranteed naturally aligned) 330
        SP ← ESI // restore SP from time of call 337
    }
    EPC ← EPC & -4 // reset the two low-order bits to zero 336
    RFE 338
  
```

FIG. 3H

340

Tapestry-to-X86 transition exception handler

// This handler is entered under the following conditions:

// 1. a native caller invokes an x86 function

// 2. a native function returns to an x86 caller

switch on XD<3:0> { 341

 XD_RET_FP: // result type is floating point

 FO/FI ← FINFLATE.de(RVDP) // X86 FP results are 80 bits

 SP ← from RXA save // discard RXA, pad, args

 FPCW ← image after FINIT & push // FP stack has 1 entry

 goto EXIT

 XD_RET_WRITEBACK: // store result to @RVA, leave RVA in eax

 RVA ← from RXA save // address of result area

 copy decode(XD<8:4>) bytes from RV0..RV3 to [RVA]

 eax ← RVA // X86 expects RVA in eax

 SP ← from RXA save // discard RXA, pad, args

 FPCW ← image after FINIT // FP stack is empty

 goto EXIT

 XD_RET_SCALAR: // result in eax:eda

 edx<31:00> ← eax<63:32> // in case result is 64 bits

 SP ← from RXA save // discard RXA, pad, args

 FPCW ← image after FINIT // FP stack is empty

 goto EXIT

 XD_CALL_HIDDEN_TEMP: // allocate 32 byte aligned hidden temp 343

 esi ← SP // stack cut back on return

 SP ← SP - 32 // allocate max size temp

 RVA ← SP // RVA consumed later by RR } 344

 LR<1:0> ← "11" // flag address for return & reload 345

 goto CALL_COMMON

 default: // remaining XD_CALL_xxx encodings

 esi ← SP // stack cut back on return

 LR<1:0> ← "10" // flag address for return 343

CALL_COMMON: 347

 interpret XD to push and/or reposition args 346

 [--SP] ← LR // push LR as return address

EXIT: } 348

 setup emulator context and profiling ring buffer pointer

 } RFE 349

 // to original target

}

FIG. 3I

350

interrupt/exception handler of Tapestry operating system:

```

// Control vectors here when a synchronous exception or asynchronous interrupt is to be
// exported to / manifested in an x86 machine.

// The interrupt is directed to something within the virtual X86, and thus there is a possibility
// that the X86 operating system will context switch. So we need to distinguish two cases:
// either the running process has only X86 state that is relevant to save, or
// there is extended state that must be saved and associated with the current machine context
// (e.g., extended state in a Tapestry library call in behalf of a process managed by X86 OS)
if execution was interrupted in the converter – EPC.ISA == X86 {
    // no dependence on extended/native state possible, hence no need to save any } 351
    goto EM86_Deliver_Interrupt( interrupt-byte )
} else if EPC.Taxi_Active {
    // A Taxi translated version of some X86 code was running. Taxi will rollback to an
    // x86 instruction boundary. Then, if the rollback was induced by an asynchronous external
    // interrupt, Taxi will deliver the appropriate x86 interrupt. Else, the rollback was induced
    // by a synchronous event so Taxi will resume execution in the converter, retriggering the
    // exception but this time with EPC.ISA == X86
    goto TAXi_Rollback( asynchronous-flag, interrupt-byte ) } 353
} else if EPC.EM86 {
    // The emulator has been interrupted. The emulator is coded to allow for such
    // conditions and permits re-entry during long running routines (e.g. far call through a gate) } 354
    // to deliver external interrupts
    goto EM86_Deliver_Interrupt( interrupt-byte )
} else {
    // This is the most difficult case - the machine was executing native Tapestry code on
    // behalf of an X86 thread. The X86 operating system may context switch. We must save
    // all native state and be able to locate it again when the x86 thread is resumed.
    // 361
    allocate a free save slot; if unavailable free the save slot with oldest timestamp and try again
    save the entire native state (both the X86 and the extended state) } 362
    save the X86 EIP in the save slot } 363
    overwrite the two low-order bits of EPC with "01" (will become X86 interrupt EIP)
    store the 64-bit timestamp in the save slot, in the X86 EBX:ECX register pair (and,
    for further security, store a redundant copy in the X86 ESI:EDI register pair) } 364
    store the a number of the allocated save slot in the X86 EAX register (and, again for
    further security, store a redundant copy in the X86 EDX register) } 365
    goto EM86_Deliver_Interrupt( interrupt-byte ) } 369
}

```

360

FIG. 3J

```

typedef struct {
    save_slot_t* newer,           // pointer to next-most-recently-allocated save slot } 379c
    save_slot_t* older;          // pointer to next-older save slot
    unsigned int64 epc;           // saved exception PC/IP
    unsigned int64 pcw;           // saved exception PCW (program control word)
    unsigned int64 registers[63]; // save the 63 writeable general registers } 356
    ...                           // other words of Tapestry context
    timestamp_t timestamp;        // timestamp to detect buffer overrun } 358
    int save_slot_ID;             // ID number of the save slot } 357
    boolean save_slot_is_full;    // full / empty flag } 359
} save_slot_t;

```

save_slot_t* save_slot_head; // pointer to the head of the queue } 379a
 save_slot_t* save_slot_tail; // pointer to the tail of the queue } 379b

} 355

system initialization
 reserve several pages of unpaged memory for save slots

FIG. 3K

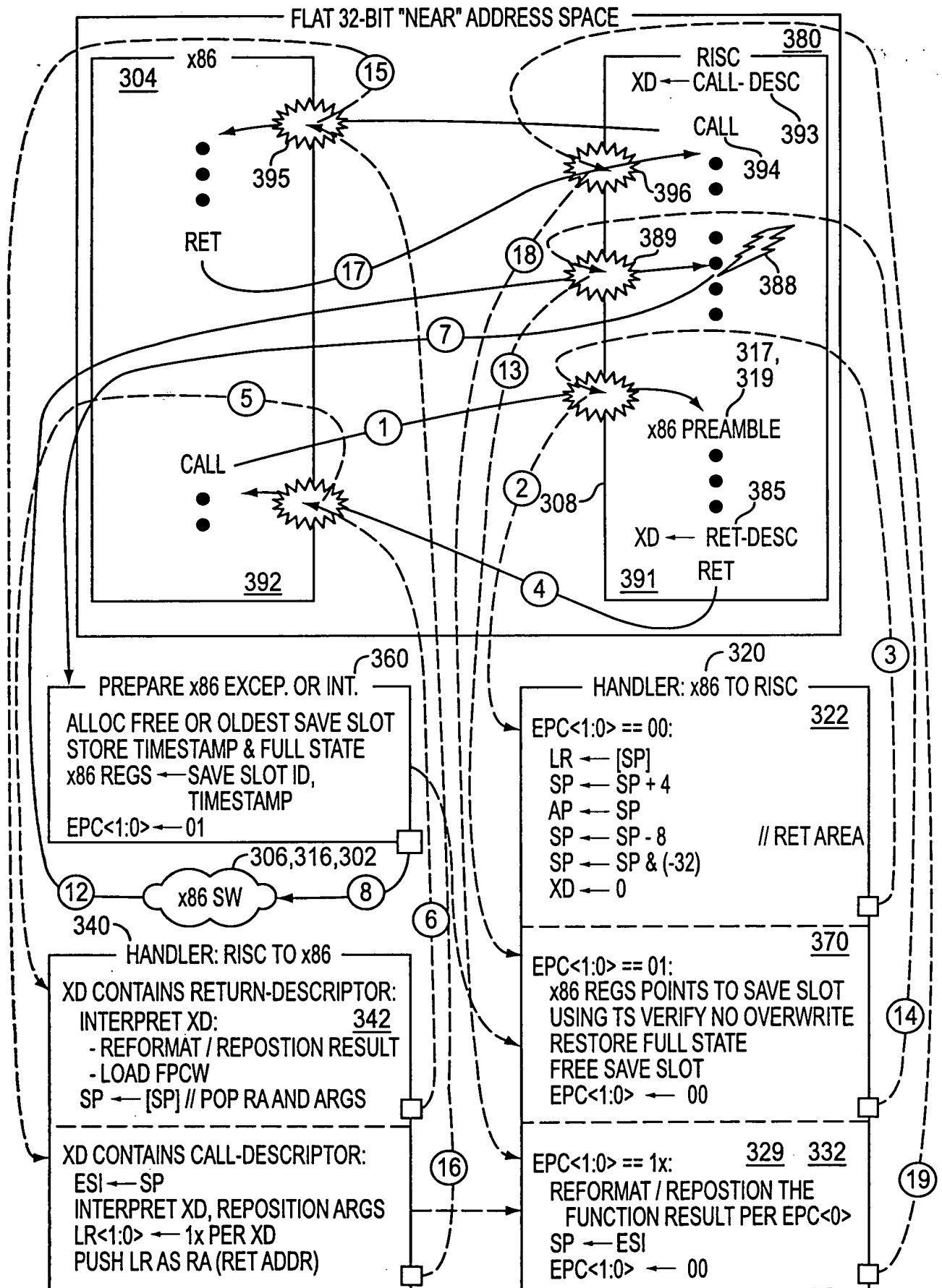


FIG. 3L

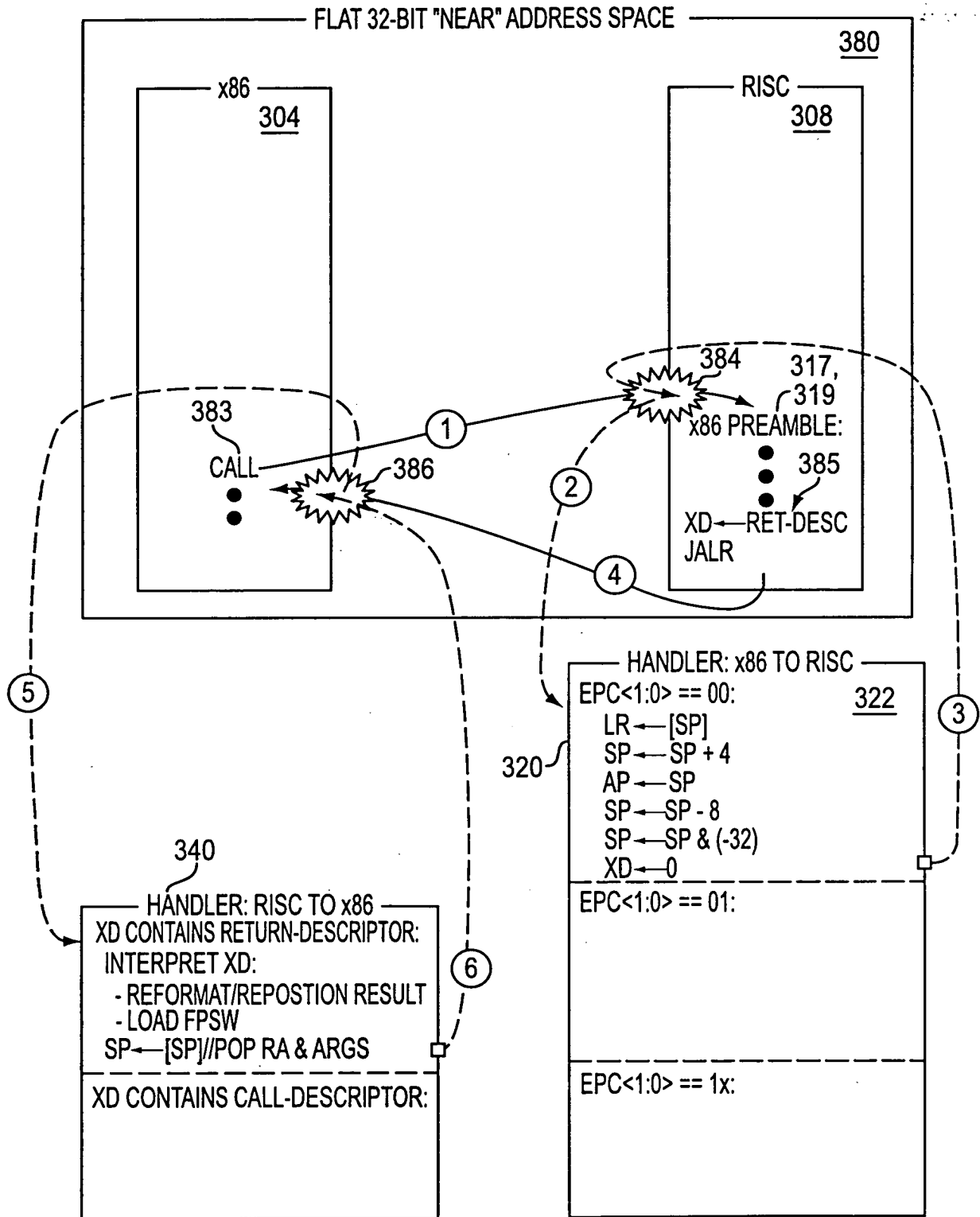


FIG. 3M

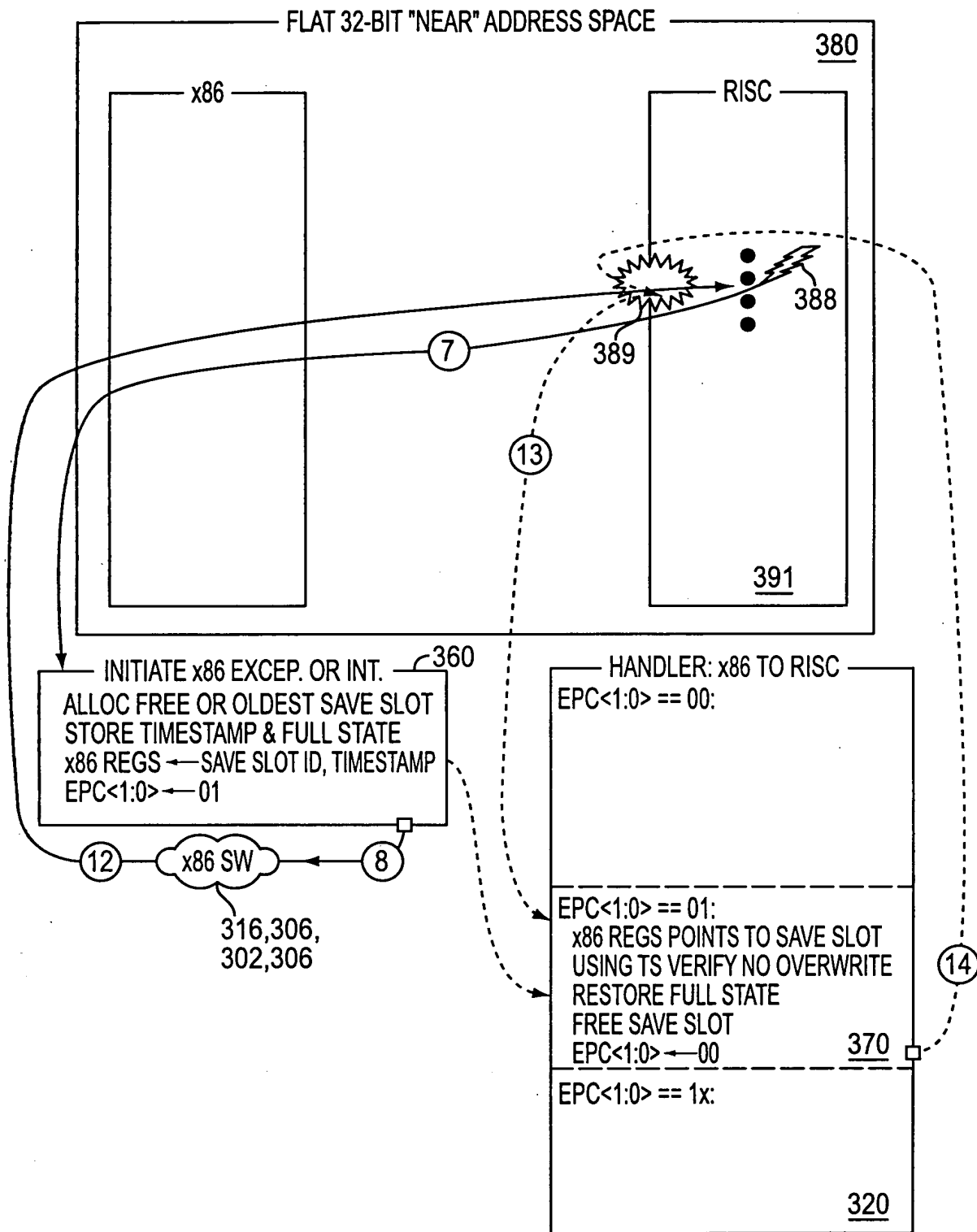


FIG. 3N

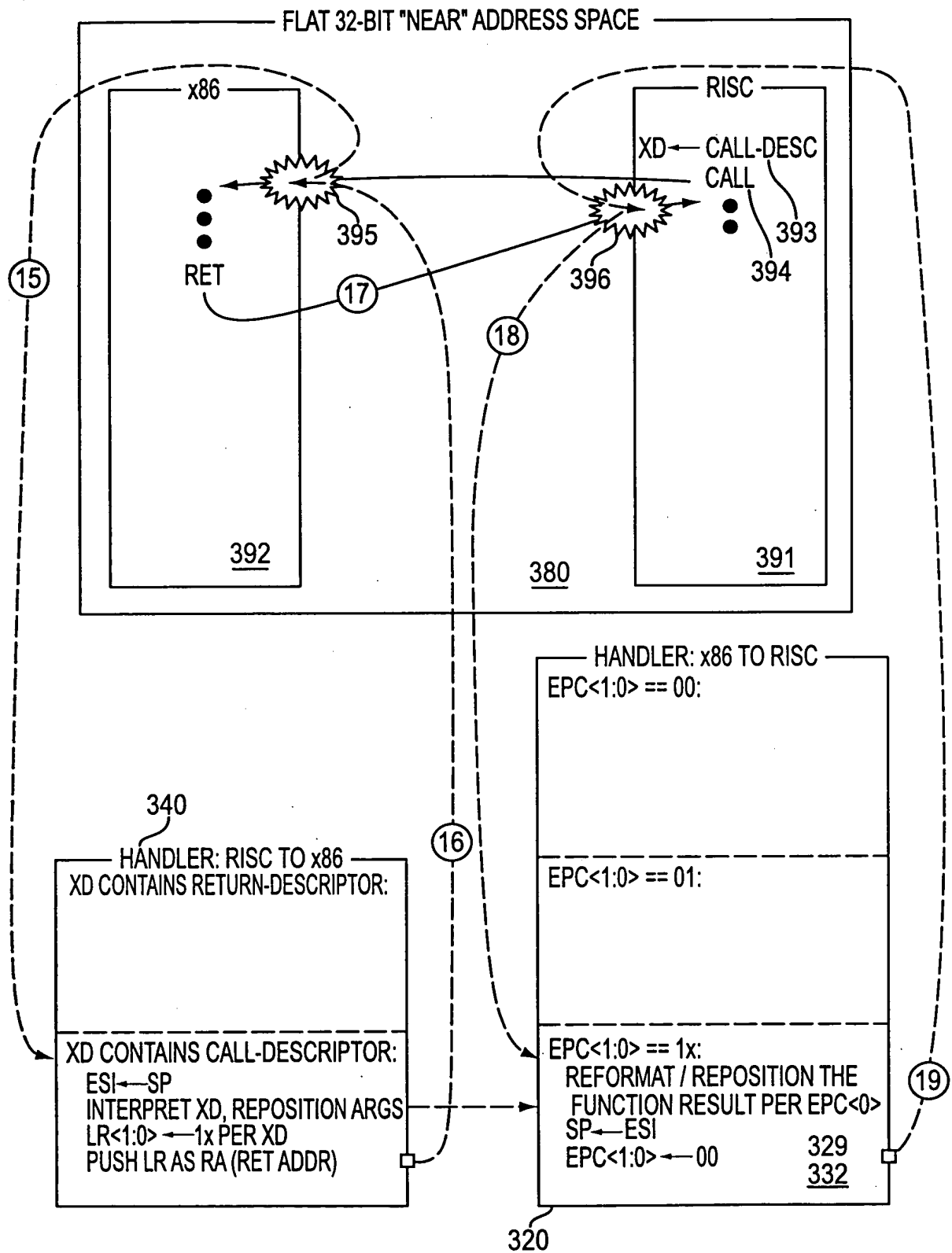
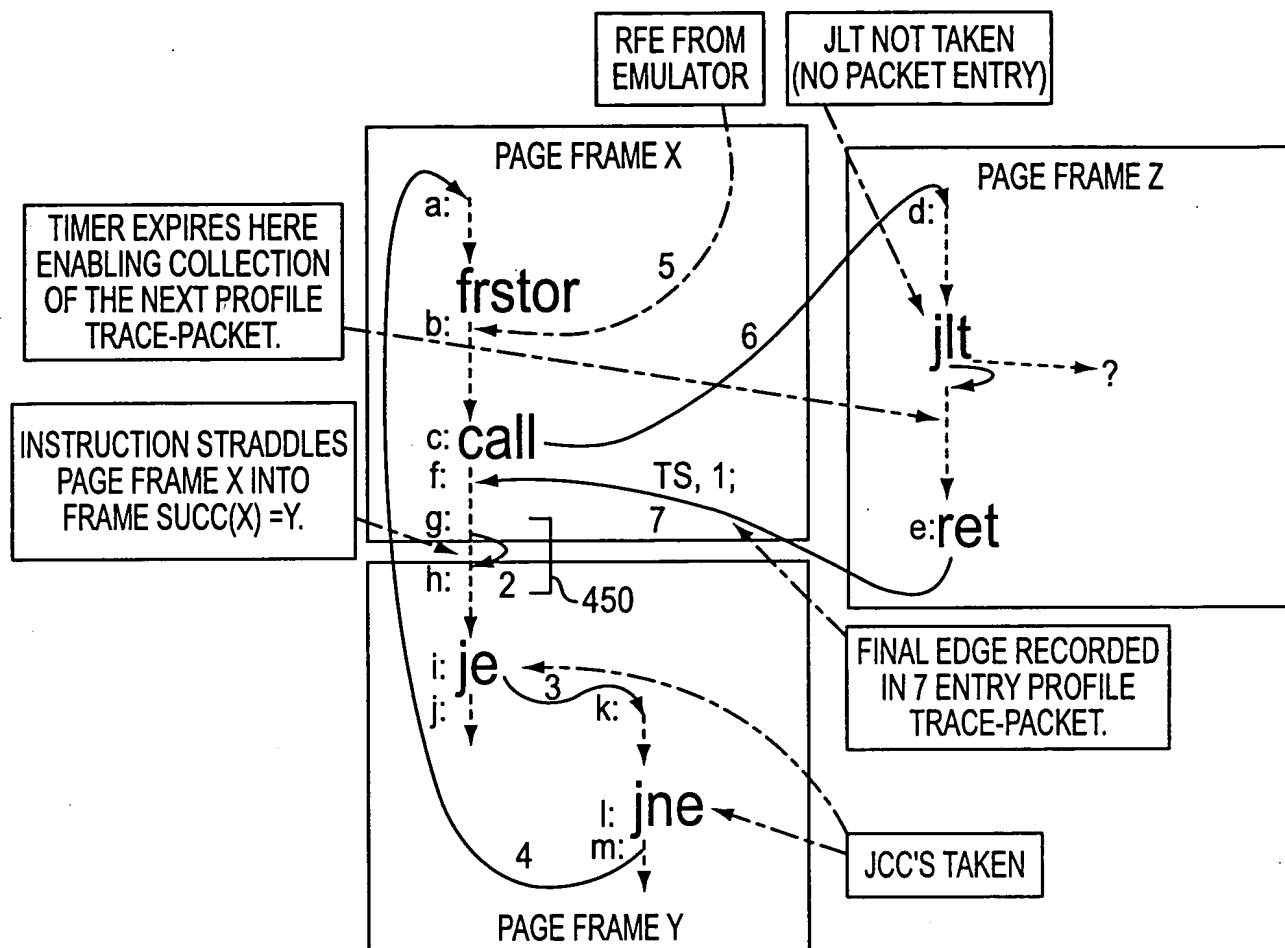


FIG. 30



7 ENTRY TRACE PACKET

ENTRY	EVENT CODE	DONE ADDR	NEXT ADDR
64 BIT TIME STAMP			
1	RET	x86 CONTEXT	phys X:f
2	NEW PAGE	phys Y:g	phys Y:h
3	JCC FORWARD	phys Y:i	phys Y:k
4	JNZ BACKWARD	phys Y:l	phys X:a
5	SEQ; ENV CHANGE	x86 CONTEXT	phys X:b
6	IP-REL NEAR CALL	phys X:c	phys Z:d
7	NEAR RET	phys Z:e	phys X:f

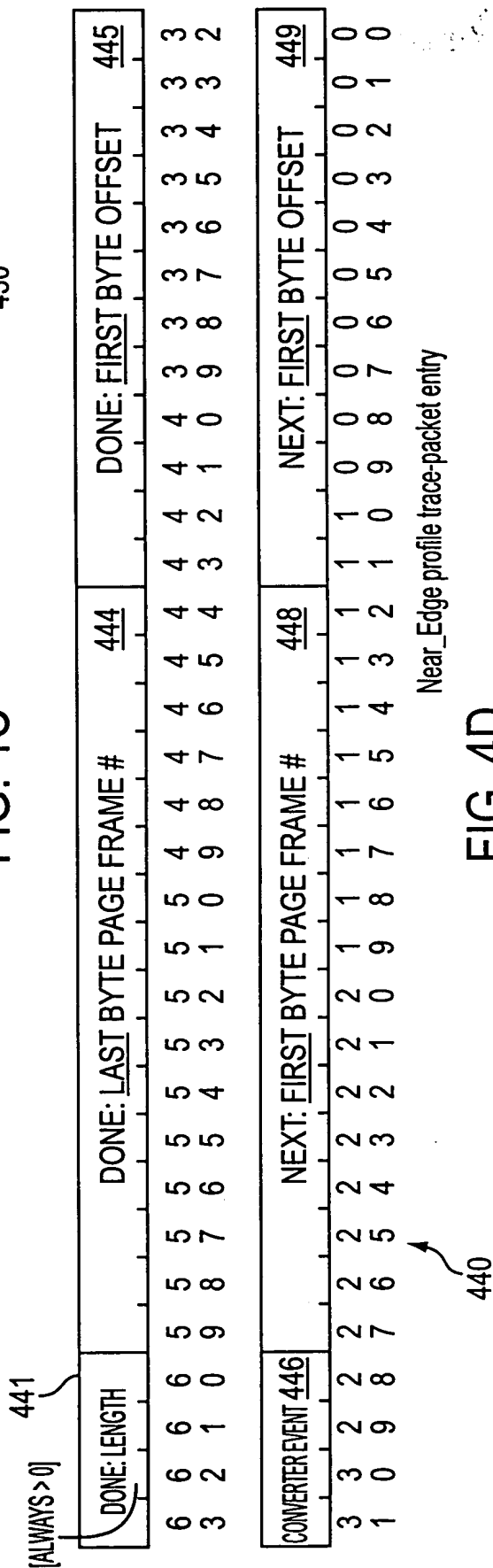
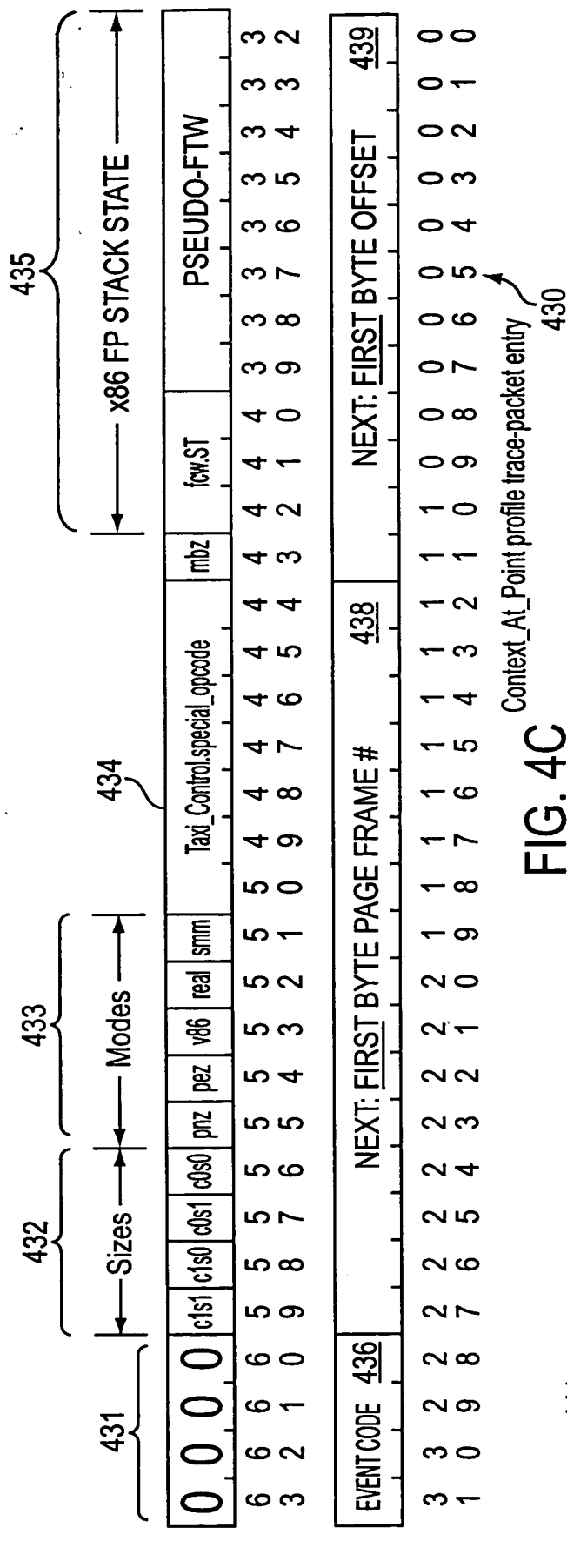
420 {

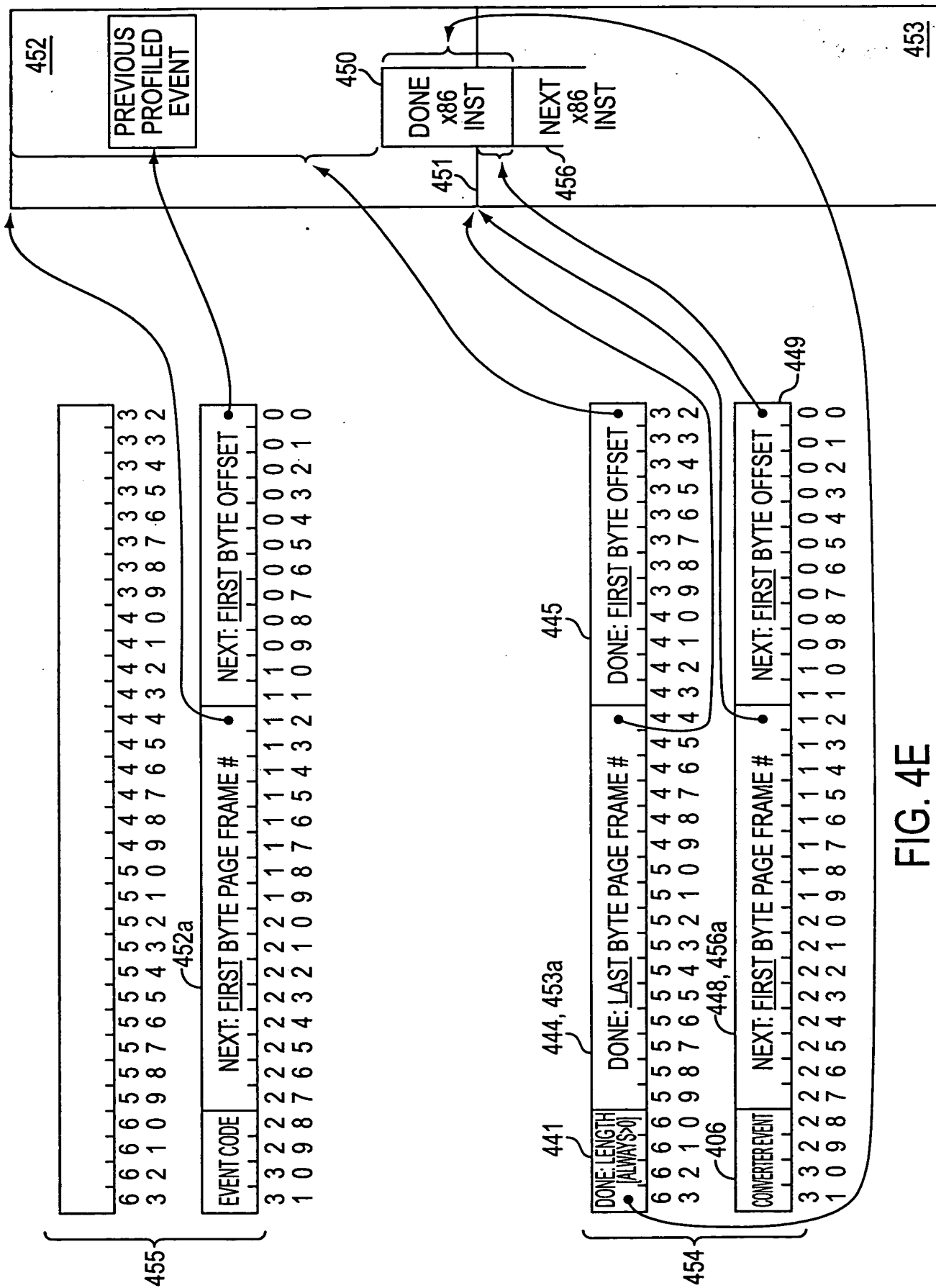
430
440, 454
440
440
430
440
440

FIG. 4A

SOURCE		414		416		418		610		612	
CODE 402		EVENT		REUSE EVENT CODE						PROBE EVENT BIT- TLB PROBE ATTRIBUTE OR EMULATOR PROBE	
412	RFE (CONTEXT AT POINT ENTRY)	0.0000	DEFAULT (x86 TRANSPARENT) EVENT, REUSE ALL CONVERTER VALUES	YES		NO				REUSE EVENT CODE	
		0.0001	SIMPLE x86 INSTRUCTION COMPLETION (REUSE EVENT CODE)	YES		NO				REUSE EVENT CODE	
		0.0010	PROBE EXCEPTION FAILED	YES		NO				REUSE EVENT CODE	
		0.0011	PROBE EXCEPTION FAILED, RELOAD PROBE TIMER	YES		NO				REUSE EVENT CODE	
		0.0100	FLUSH EVENT	NO	NO	NO	NO			.	
		0.0101	SEQUENTIAL; EXECUTION ENVIRONMENT CHANGED - FORCE EVENT	NO	YES	NO	NO			.	
		0.0110	FAR RET	NO	YES	YES	NO			.	
		0.0111	IRET	NO	YES	NO	NO			.	
		0.1000	FAR CALL	NO	YES	YES	YES			FAR CALL	
		0.1001	FAR JMP	NO	YES	YES	NO			.	
		0.1010	SPECIAL; EMULATOR EXECUTION, SUPPLY EXTRA INSTRUCTION DATA ^a	NO	YES	NO	NO			.	
		0.1011	ABORT PROFILE COLLECTION	NO	NO	NO	NO			.	
		0.1100	x86 SYNCHRONOUS/ASYNCHRONOUS INTERRUPT W/PROBE (GRP 0)	NO	YES	YES	YES			EMULATOR PROBE	
		0.1101	x86 SYNCHRONOUS/ASYNCHRONOUS INTERRUPT (GRP 0)	NO	YES	YES	NO			.	
		0.1110	x86 SYNCHRONOUS/ASYNCHRONOUS INTERRUPT W/PROBE (GRP 1)	NO	YES	YES	YES			EMULATOR PROBE	
		0.1111	x86 SYNCHRONOUS/ASYNCHRONOUS INTERRUPT (GRP 1)	NO	YES	YES	NO			.	
404	CONVERTER (NEAR EDGE ENTRY)	1.0000	IP-RELATIVE JNZ FORWARD (OPCODE: 75, OF 85)	NO	YES	YES	NO			.	
		1.0001	IP-RELATIVE JNZ BACKWARD (OPCODE: 75, OF 85)	NO	YES	YES	YES			JNZ	
		1.0010	IP-RELATIVE CONDITIONAL JUMP FORWARD - (JCC, JCXZ, LOOP)	NO	YES	YES	NO			.	
		1.0011	IP-RELATIVE CONDITIONAL JUMP BACKWARD - (JCC, JCXZ, LOOP)	NO	YES	YES	YES			COND JUMP	
		1.0100	IP-RELATIVE, NEAR JMP FORWARD (OPCODE: E9, EB)	NO	YES	YES	NO			.	
		1.0101	IP-RELATIVE, NEAR JMP BACKWARD (OPCODE: E9, EB)	NO	YES	YES	YES			NEAR JUMP	
		1.0110	RET/RET IMM16 (OPCODE C3, C2 M)	NO	YES	YES	NO			.	
		1.0111	IP-RELATIVE, NEAR CALL (OPCODE: E8)	NO	YES	YES	YES			NEAR CALL	
		1.1000	REPE/REPNE CMPS/SCAS (OPCODE: A6, A7, AE, AF)	NO	YES	NO	NO			.	
		1.1001	REP MOVS/STOS/LDS (OPCODE: A4, A5, AA, AB, AC, AD)	NO	YES	NO	NO			.	
		1.1010	INDIRECT NEAR JMP (OPCODE: FF 14)	NO	YES	YES	NO			.	
		1.1011	INDIRECT NEAR CALL (OPCODE: FF 12)	NO	YES	YES	YES			NEAR CALL	
		1.1100	LOAD FROM I/O MEMORY (TLB.ASI != 0) (NOT USED IN T1)	NO	YES	NO	NO			.	
		1.1101	AVAILABLE FOR EXPANSION	NO	NO	NO	NO			.	
		1.1110	DEFAULT CONVERTER EVENT; SEQUENTIAL 406	NO	NO	NO	NO			.	
		1.1111	NEW PAGE (INSTRUCTION ENDS ON LAST BYTE OF A PAGE FRAME OR STRADDLES ACROSS A PAGE FRAME BOUNDARY) 408	NO	YES	NO	NO			.	

FIG. 4B





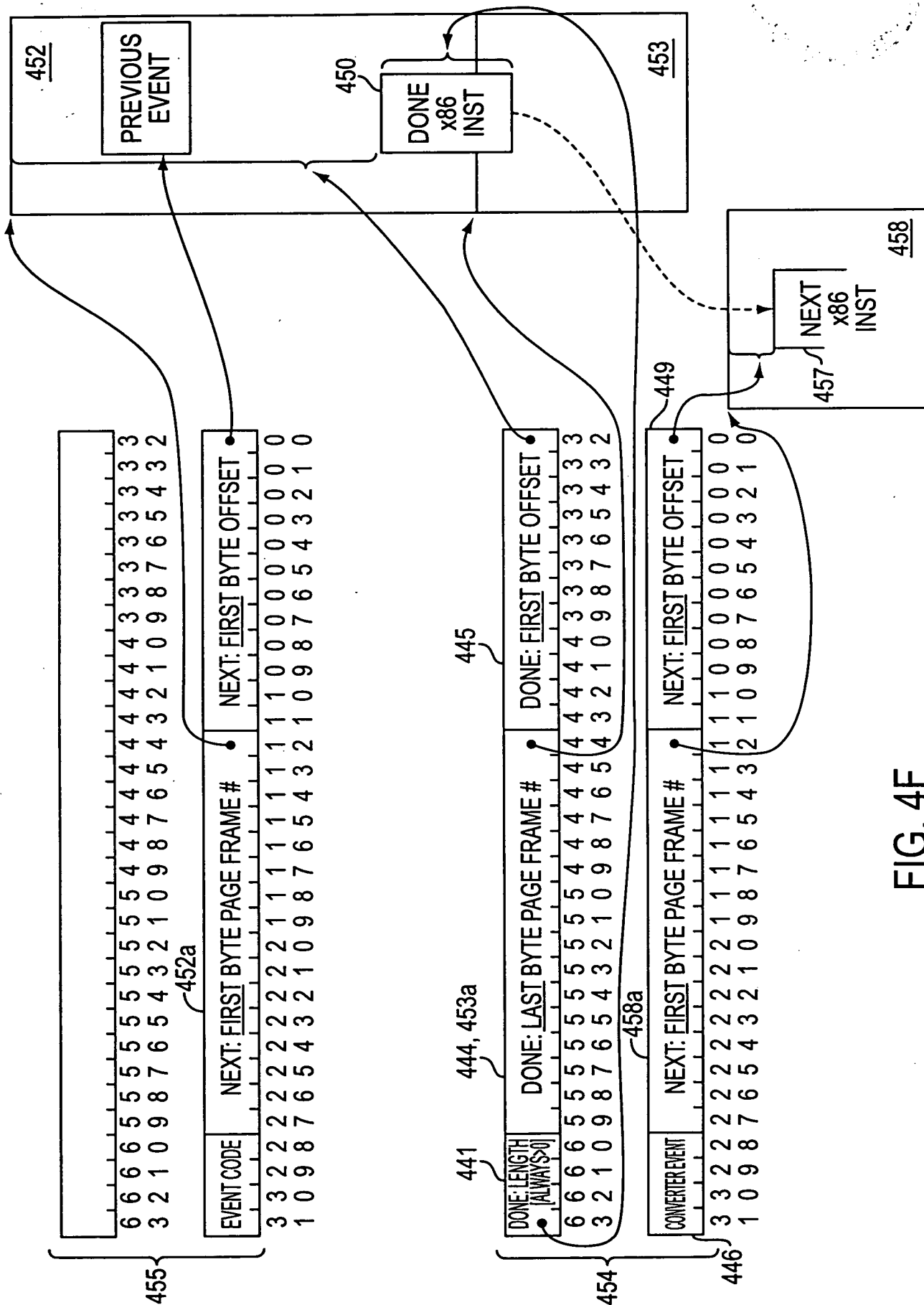
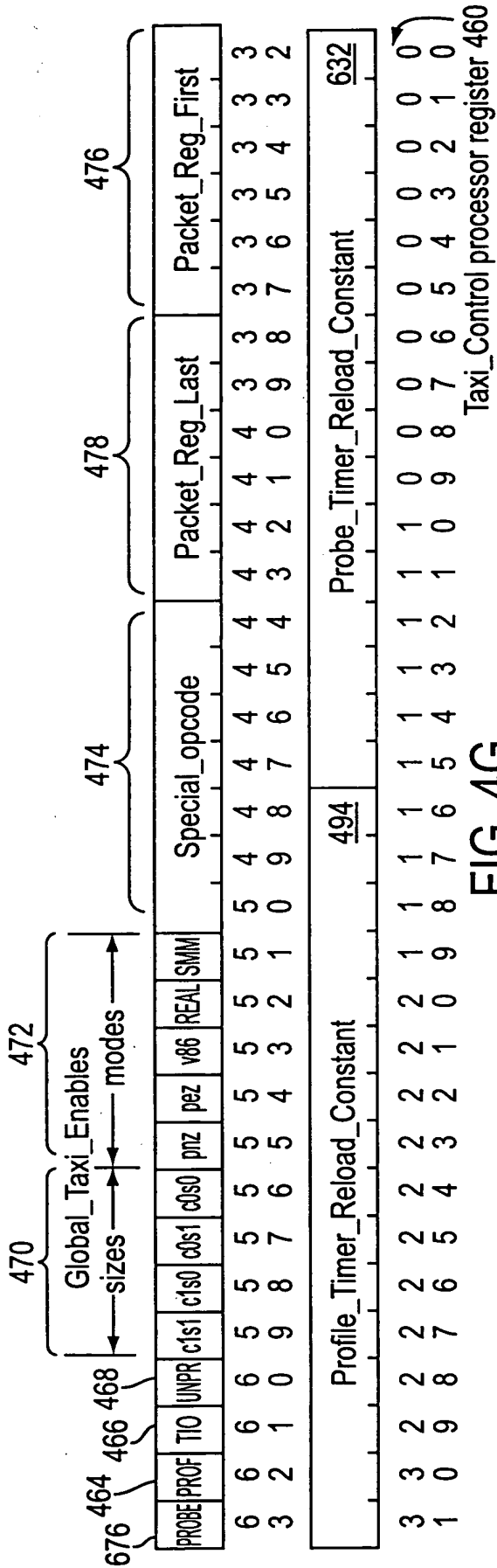


FIG. 4F



EVENTS
 pe_{init} = "initiate packet" profile event
 pe_{init} = non-"initiate packet" profile event
 pe = any profile event
 te = timer expiry
 ap = abort packet

STATE VARIABLES
 PR = Profile_Request flag
 PA = Profile_Active flag

RULES
 te EVENT:
 PR ← 1
 PR & PA & pe_{init}:
 PR ← 0
 PA ← 1
 Init Packet_Reg
 Save Timestamp
 Log Event (CAP)
 Full Packet? / Packet_Reg++

PA & pe:
 Log Event (CAP or NE)
 Full Packet? / Packet_Reg++

FULL PACKET:
 PA ← 0
 profile exception
 ap EVENT:
 PA ← 0

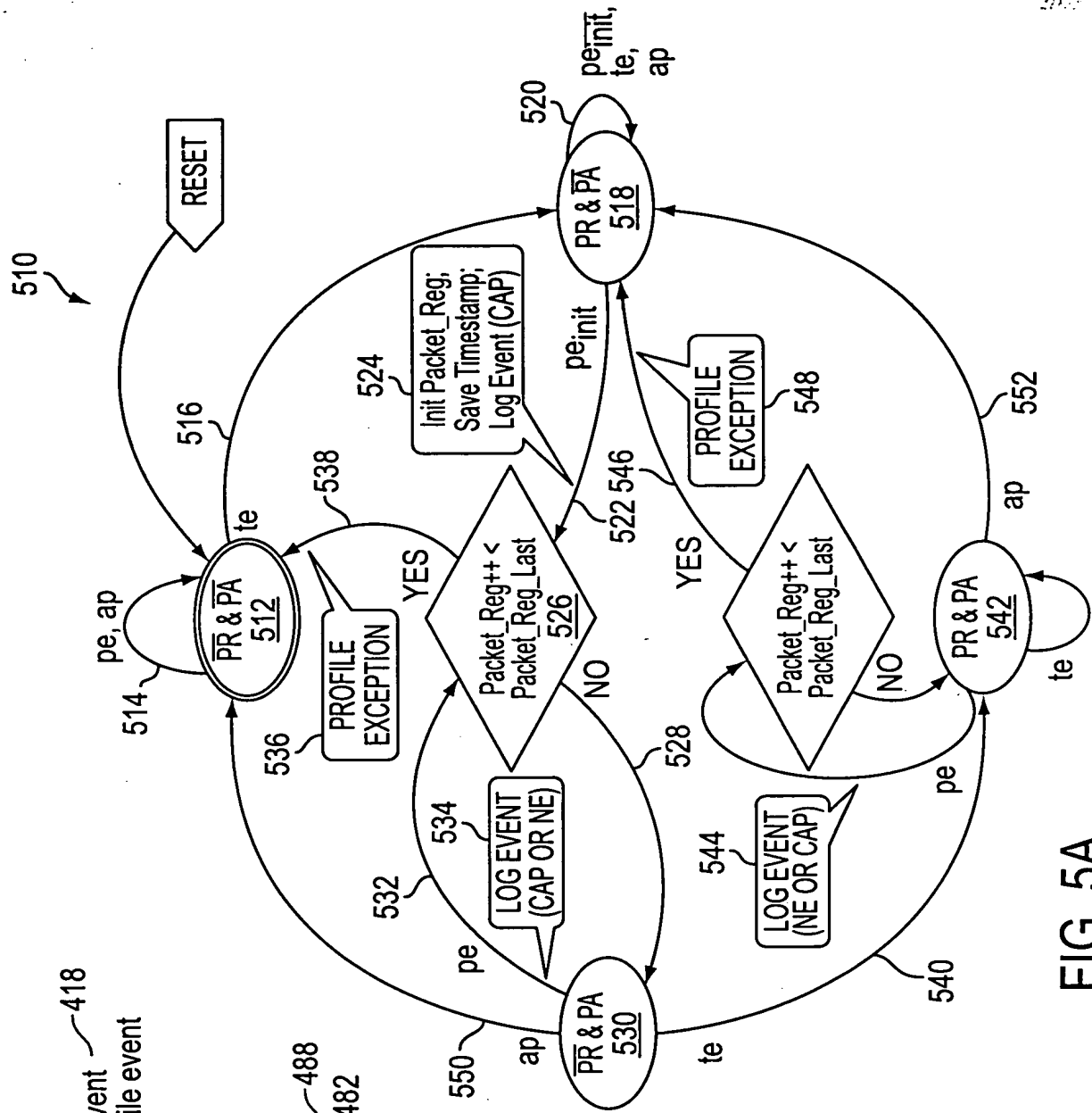
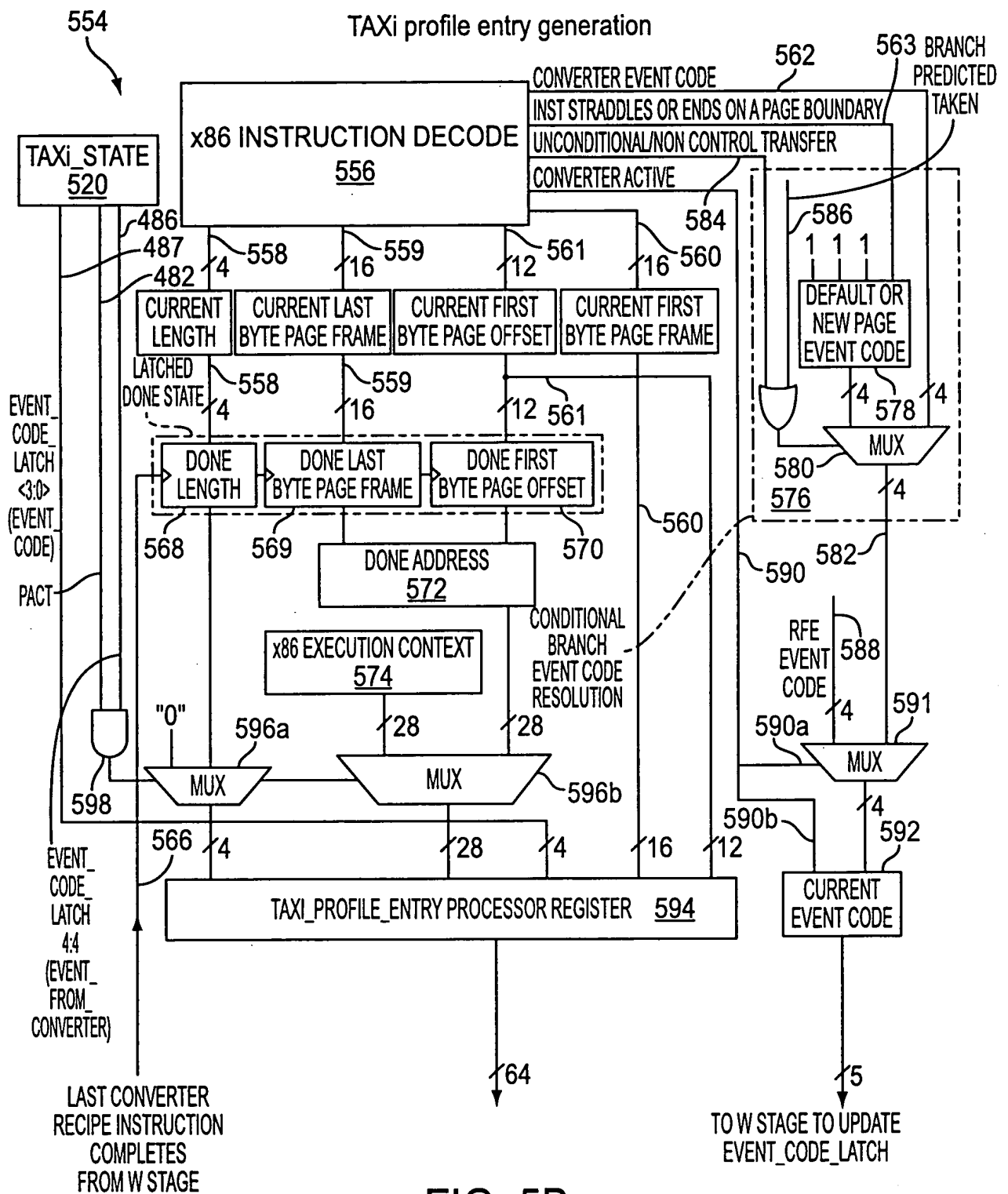


FIG. 5A



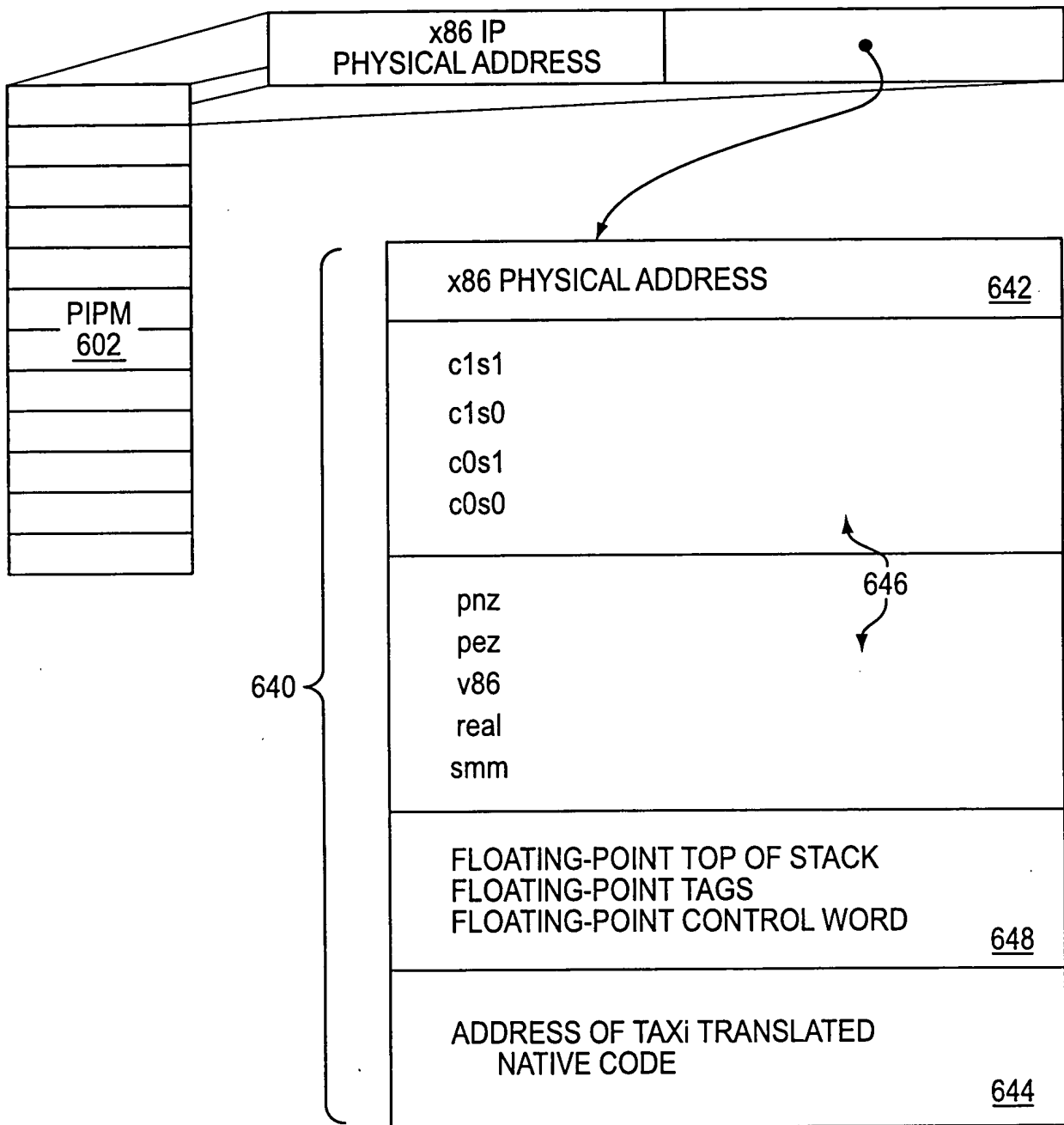


FIG. 6A

EVENT CODE FROM RFE RESTARTING CONVERTER
OR MAPPING OF CONVERTER'S x86 OPCODE

RFE OR PREVIOUS CONVERTER CYCLE

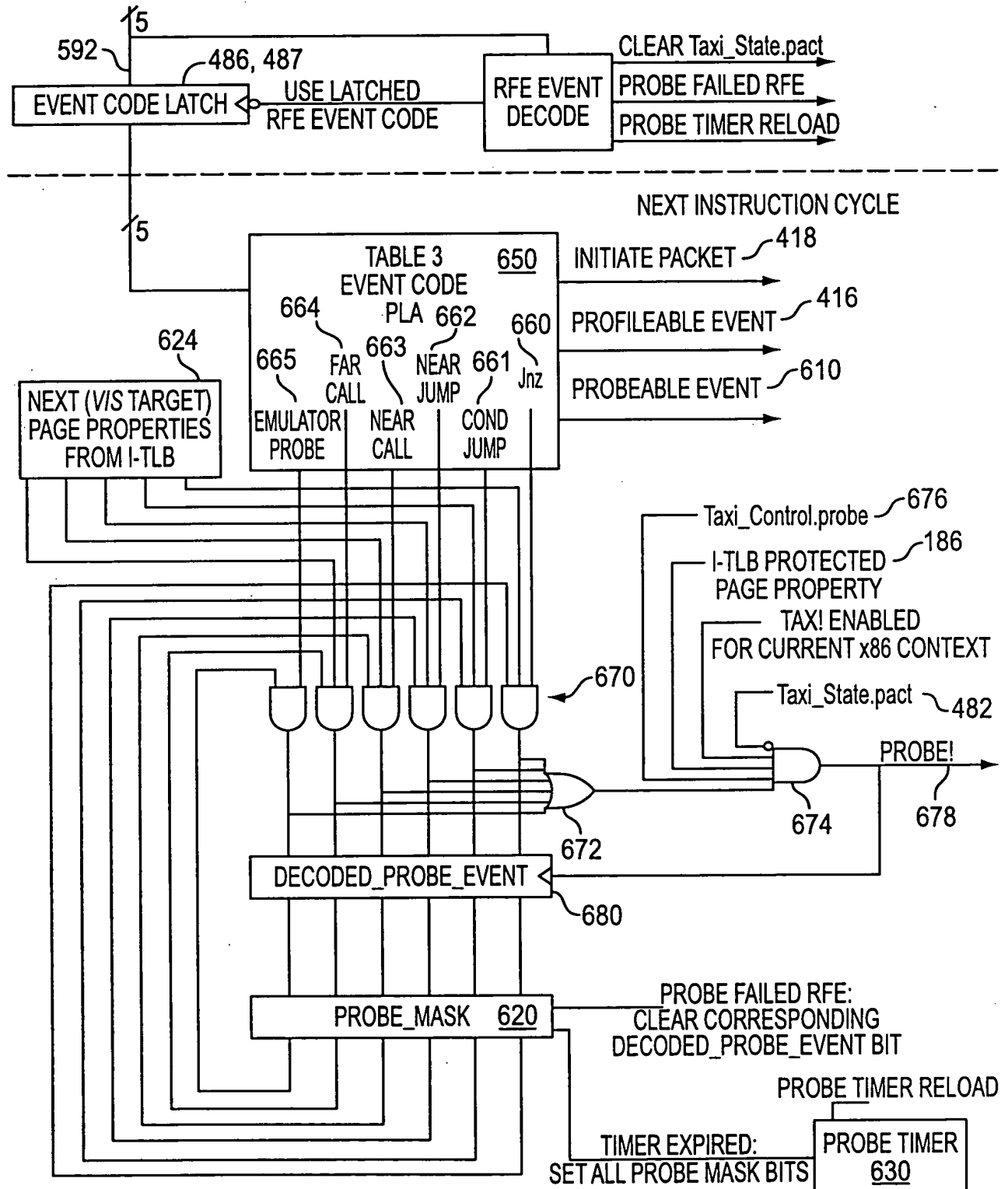


FIG. 6B

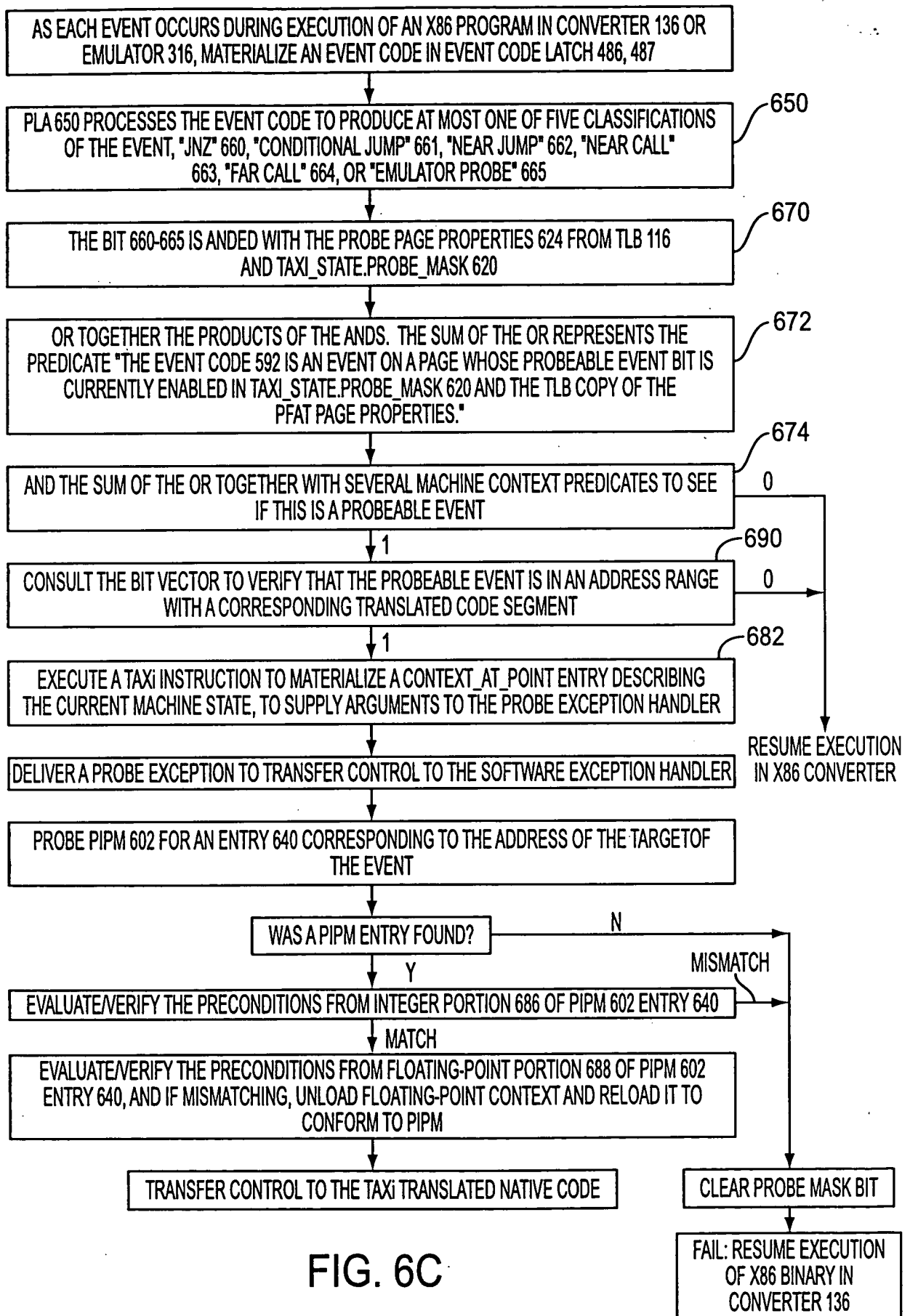
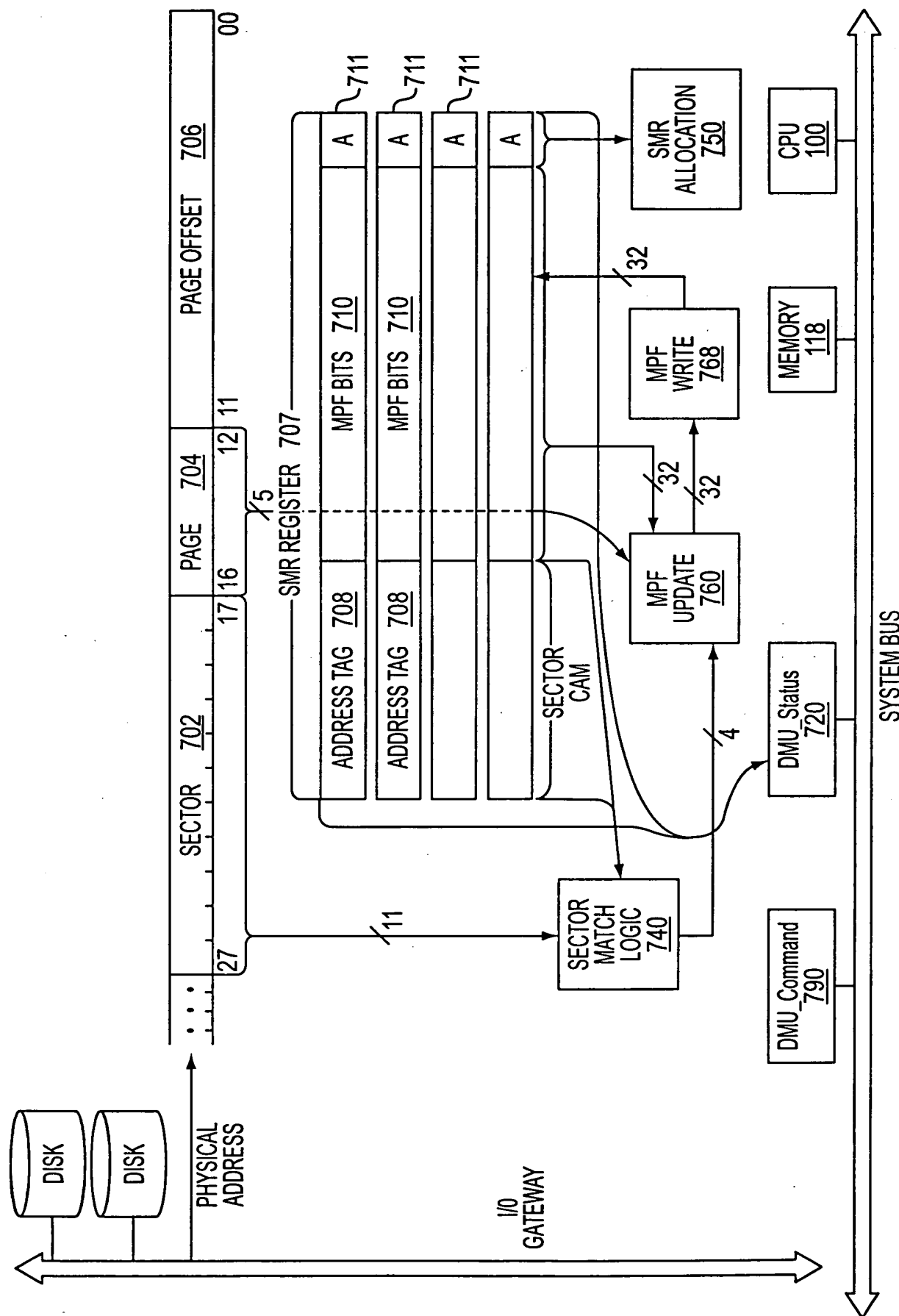


FIG. 6C



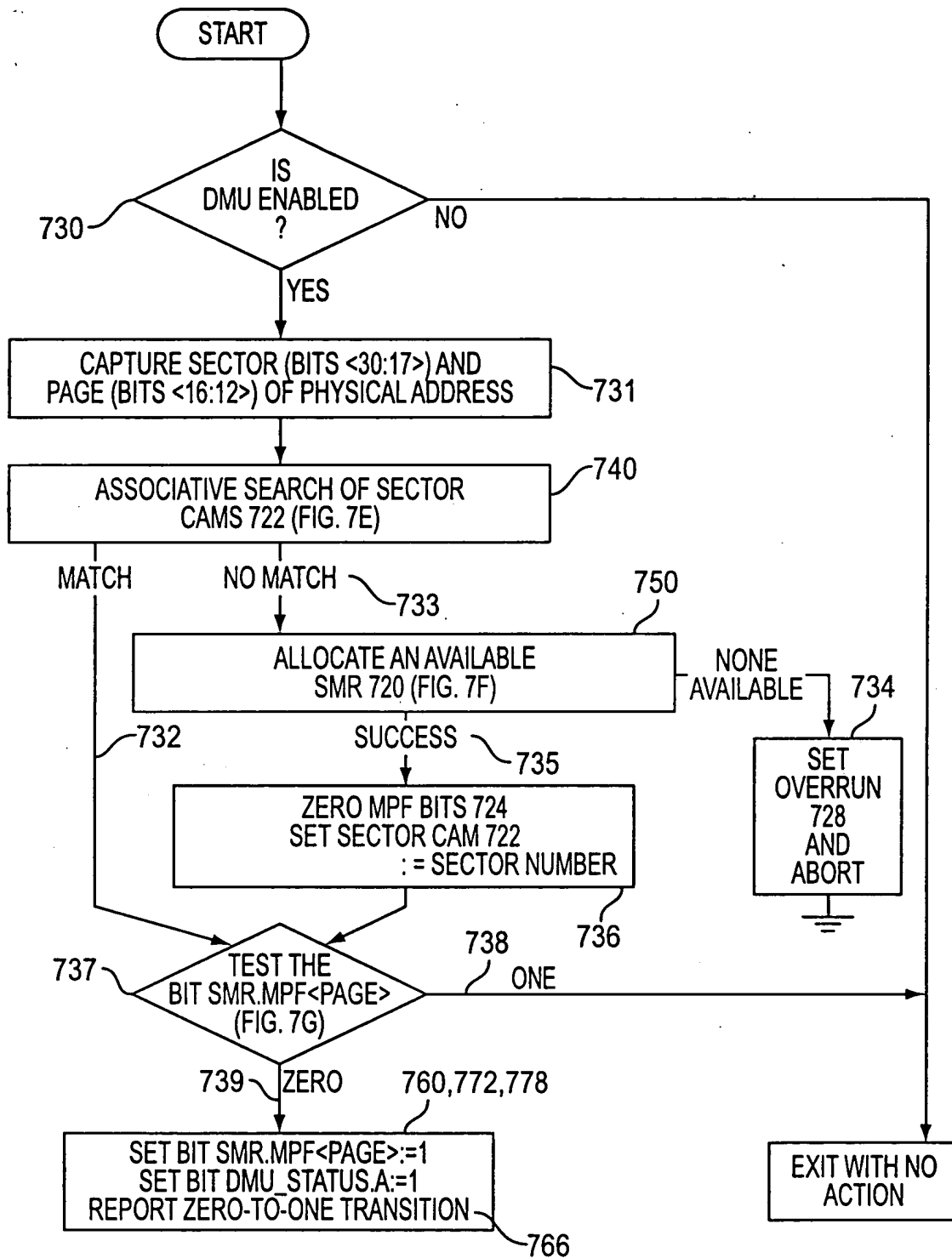
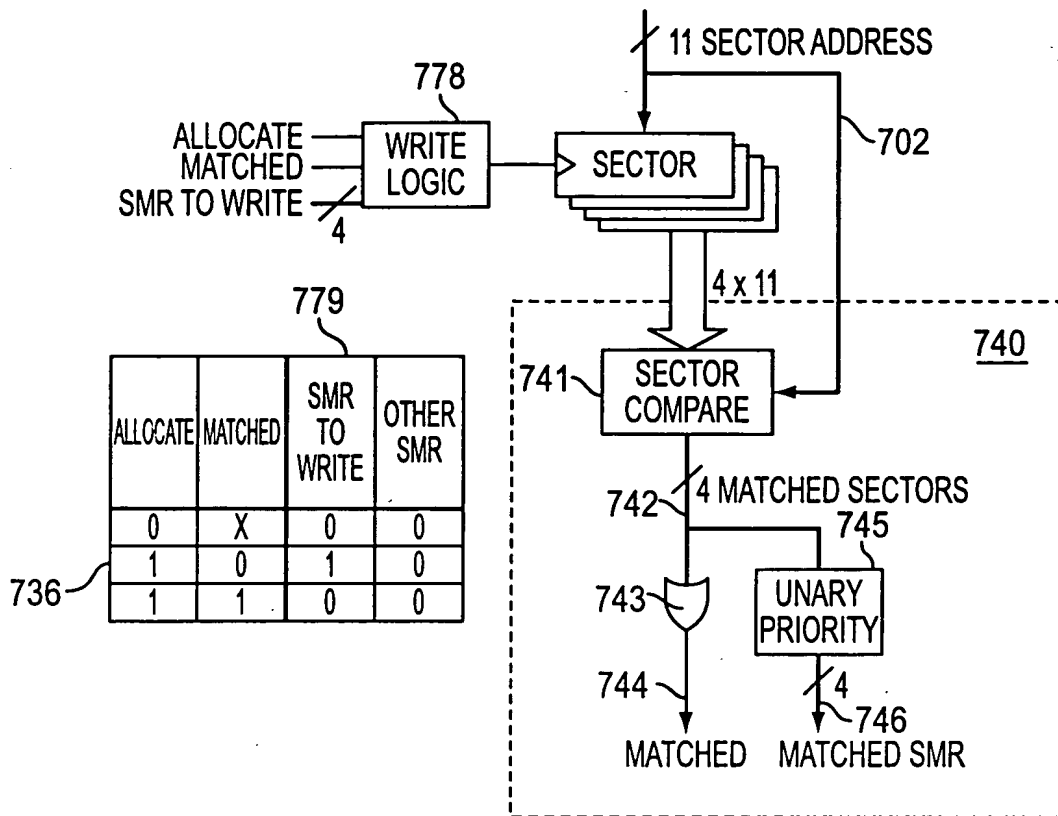
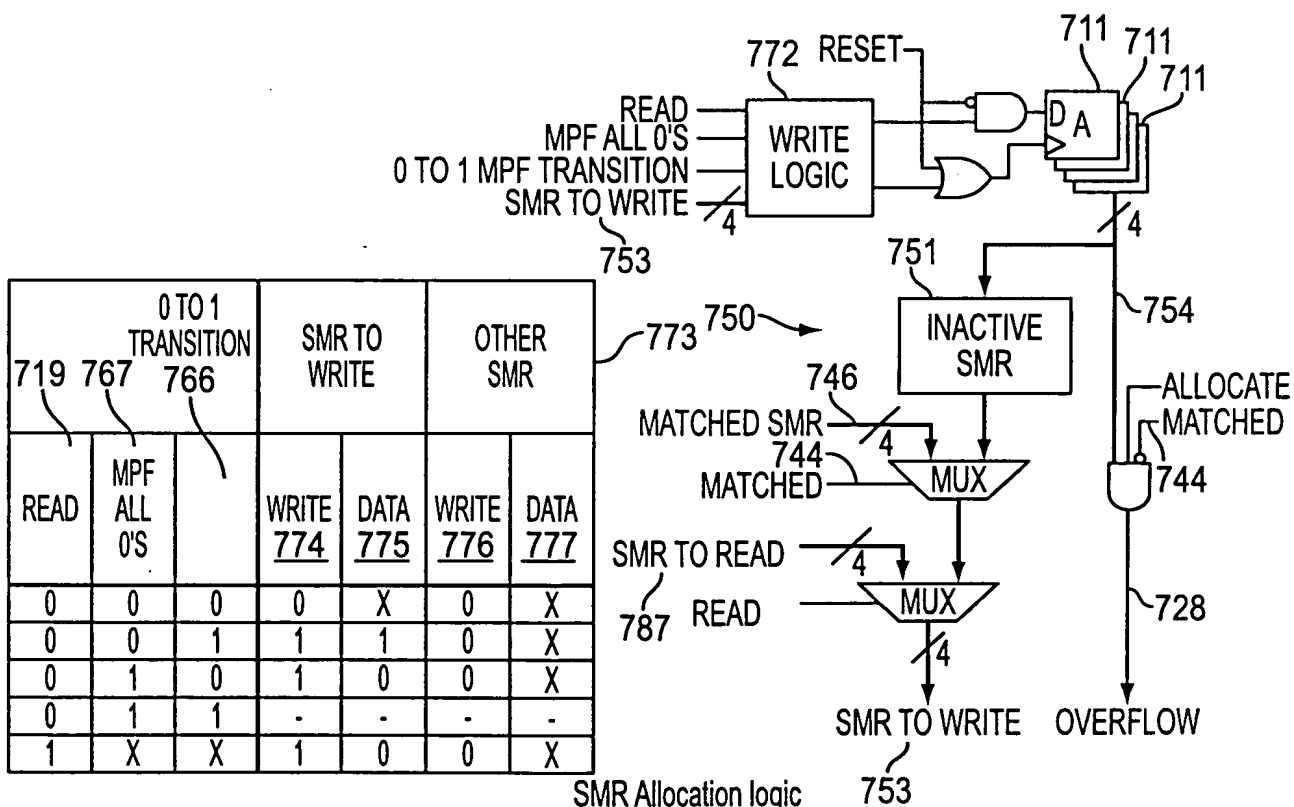


FIG. 7D



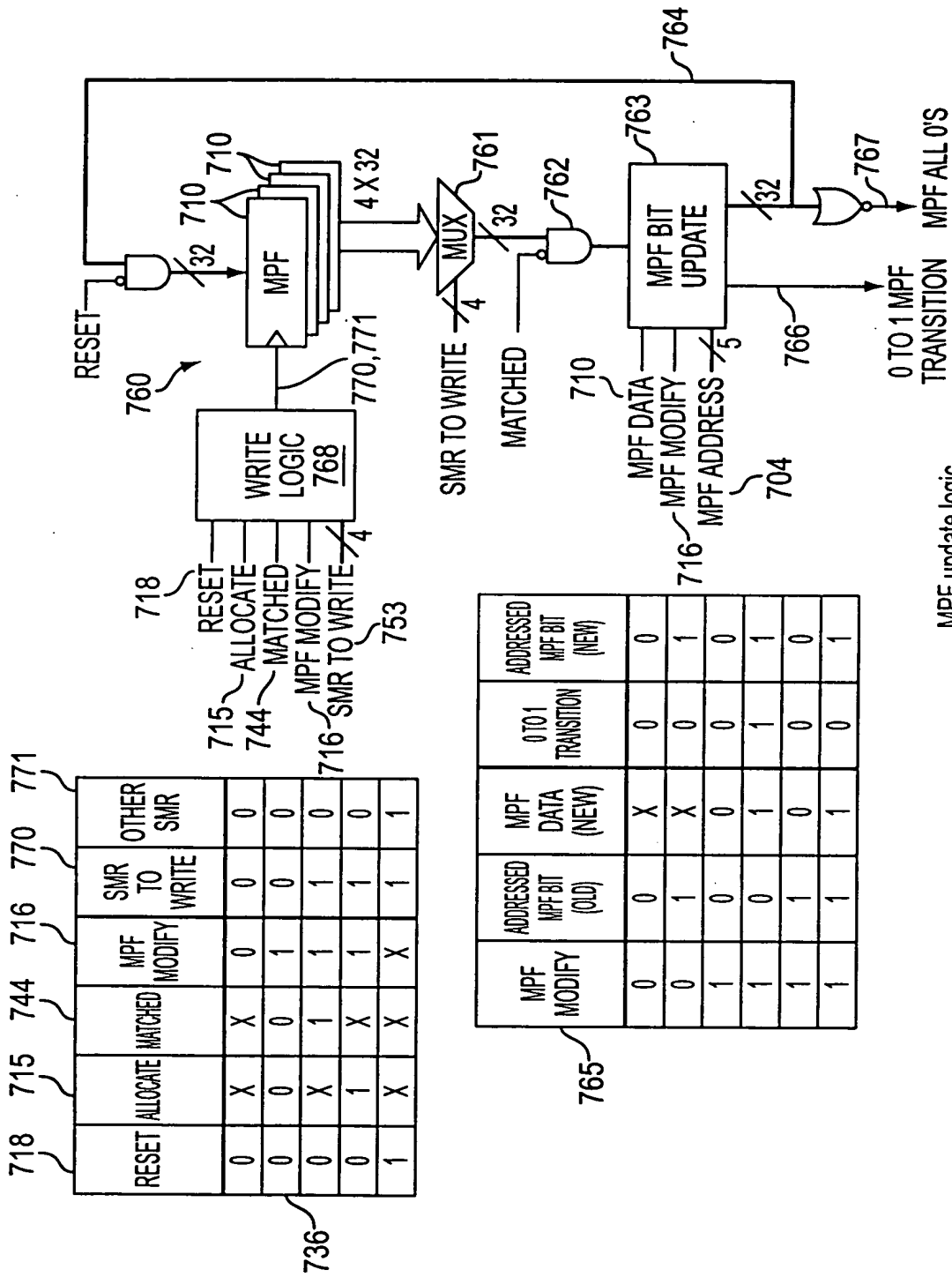
Sector match logic

FIG. 7E



SMR Allocation logic

FIG. 7F



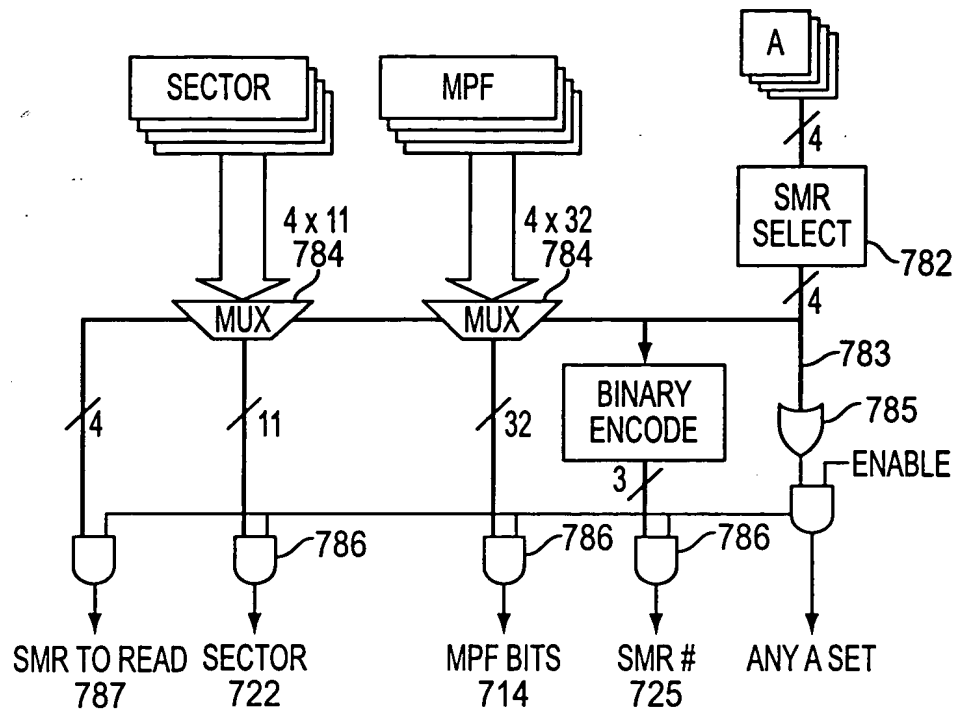


FIG. 7H

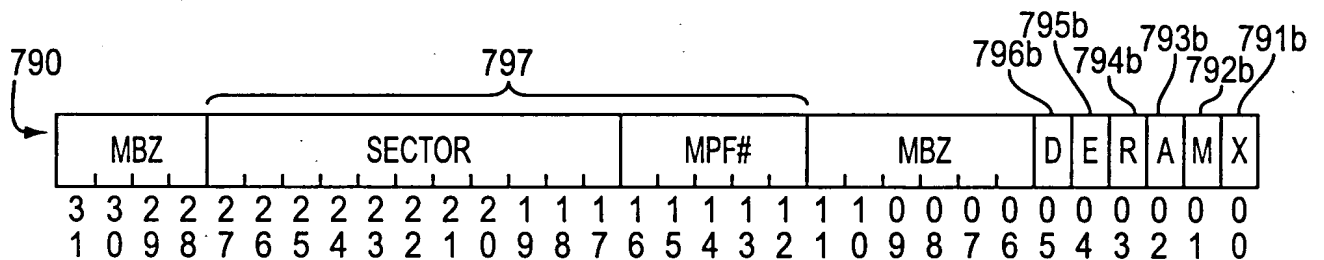
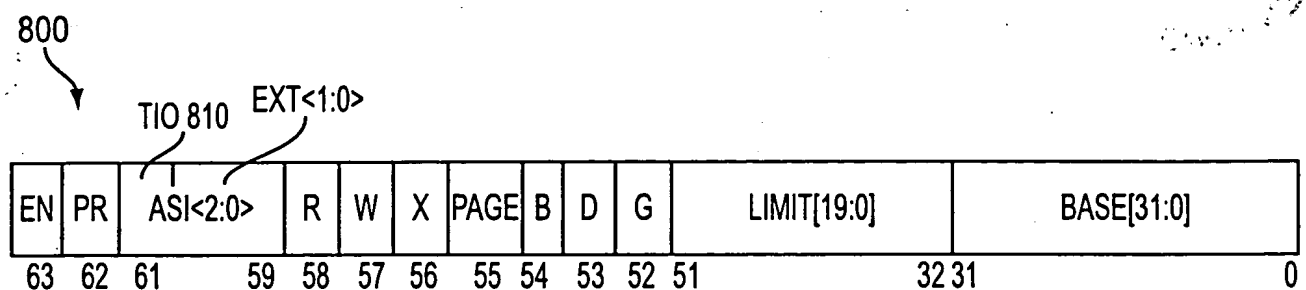


FIG. 7I

COMMAND BIT	BIT POSITION	MEANING
D	5	DISABLE MONITORING OF DMA WRITES BY CLEARING THE DMU ENABLE FLAG
E	4	ENABLE MONITORING OF DMA WRITES BY SETTING THE DMU ENABLE FLAG
R	3	RESET ALL SMRS: CLEAR ALL A AND MPF BITS AND CLEAR THE DMU OVERRUN FLAG
A	2	ALLOCATE AN INACTIVE SMR ON A FAILED SEARCH
M	1	ALLOW MPF MODIFICATIONS
X	0	NEW MPF BIT VALUE TO RECORD ON SUCCESSFUL SEARCH (OR ALLOCATION)

M	X	ACTION
0	-	INHIBIT MODIFICATION OF THE MPF BIT
1	0	CLEAR THE CORRESPONDING MPF BIT
1	1	SET THE CORRESPONDING MPF BIT

FIG. 7J



SIZE	BIT(S)	NAME	FUNCTION
1	63	SEG.EN	ENABLES SEGMENT LIMIT/PROTECTION CHECKING
1	62	SEG.PR	CHOOSES WHICH PROTECTION BITS TO USE FOR PAGE TABLE PROTECTION - (0 MEANS PSW.UK OR 1 MEANS MISC.UK)
3	61:59	SEG.AS	ADDRESS SPACE (ONLY USED WHEN SEG.PAGE IS 0)
		SEG.TIO, SEG.EXT	ADDRESS SPACE EXTENSION (ONLY USED WHEN SEG.PAGE IS 1)
3	58:56	SEG.RWX	READ/WRITE/EXECUTE '1' MEANS ENABLED - ALL 000 MEANS IT'S AN INVALID SEGMENT
1	55	SEG.PAGE	ENABLES THE PAGING SYSTEM -- (TRANSLATION AND CHECKING)
1	54	SEG.B	SEGMENT SIZE (1 MEANS 32-BIT, 0 MEANS 16-BIT)
1	53	SEG.D	SEGMENT DIRECTION (0 MEANS EXPAND UP)
1	52	SEG.G	SIZE OF LIMIT (1 MEANS IT'S IN 4k PAGES)
20	51:32	SEG.LIMIT	SEGMENT LIMIT
32	31:0	SEG.BASE	SEGMENT BASE

FIG. 8A

AT CODE GENERATION TIME:

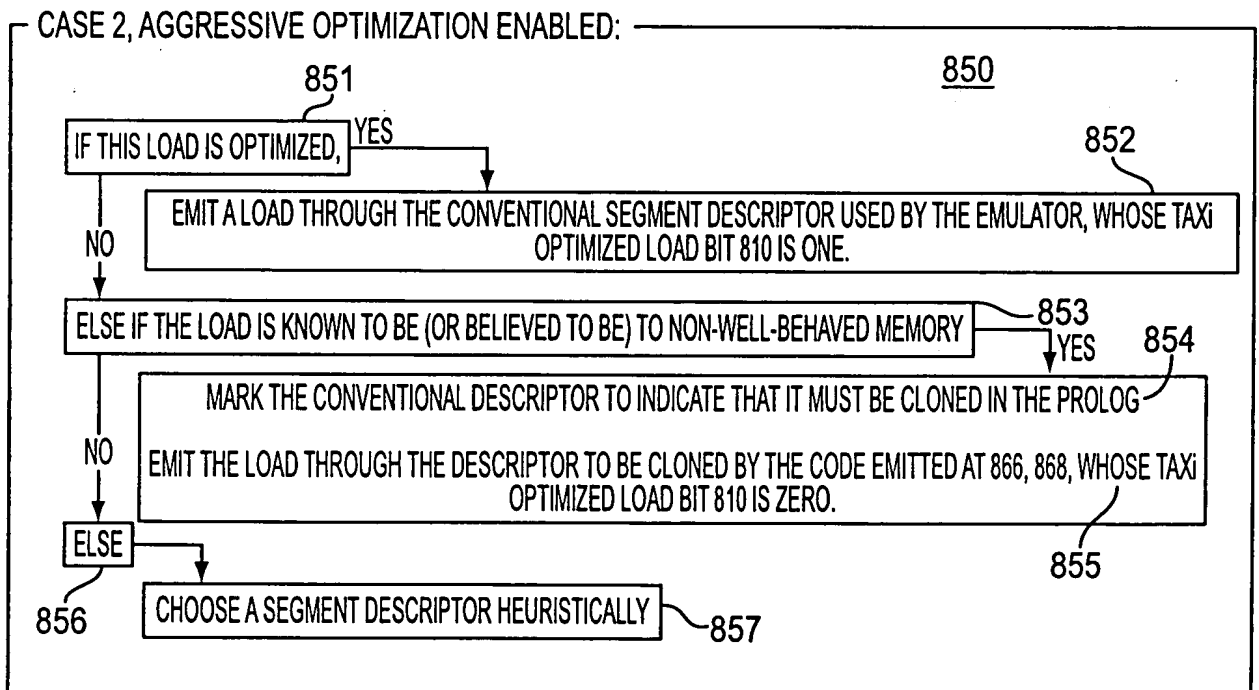
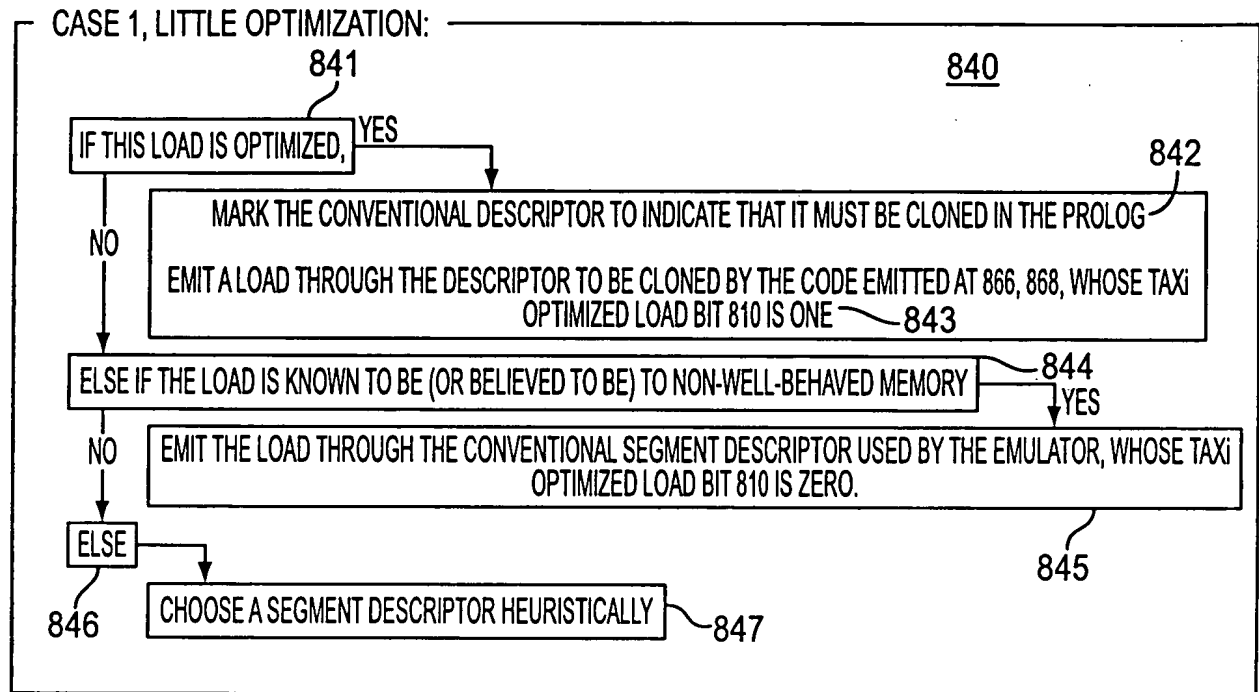


FIG. 8B

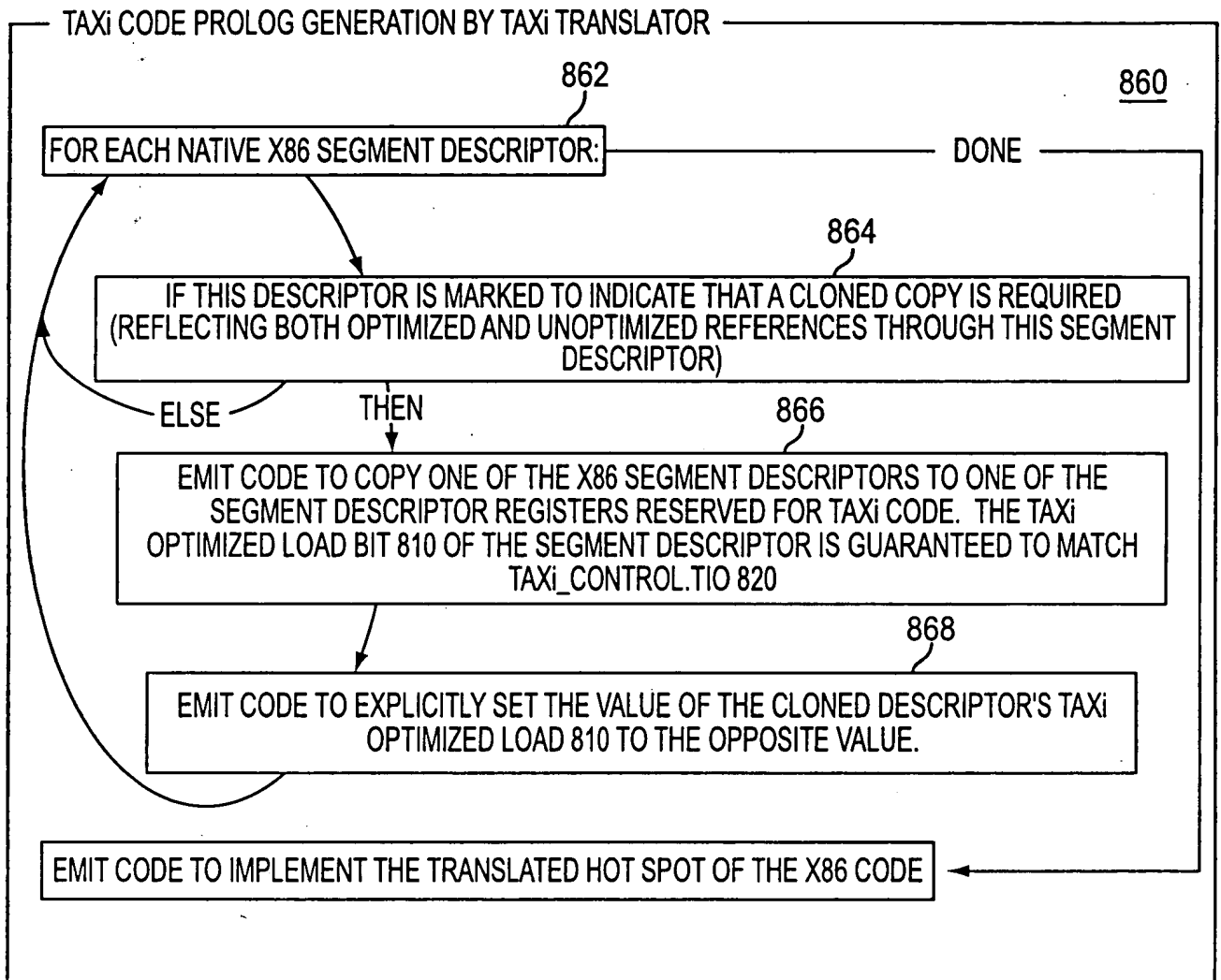


FIG. 8C